Erasmus+

GAMEHUB University Enterprises Cooperation in Game Industry in Ukraine

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРНЕТИКИ ТА СИСТЕМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

ЖАРІКОВА М.В.

ЕЛЕКТРОННИЙ НАВЧАЛЬНИЙ ПОСІБНИК

«Розробка ігрових web-додатків»

Для підготовки студентів
на першому (бакалаврському) рівні вищої освіти,
галузі знань 12 «Інформаційні технології»,
спеціальності 121 «Інженерія програмного забезпечення»,
освітньо-професійної програми «Програмна інженерія»

ХЕРСОН-2018

Erasmus+

GAMEHUB University Enterprises Cooperation in Game Industry in Ukraine

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРНЕТИКИ ТА СИСТЕМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

ЕЛЕКТРОННИЙ НАВЧАЛЬНИЙ ПОСІБНИК

«Розробка ігрових web-додатків»

Для підготовки студентів
на першому (бакалаврському) рівні вищої освіти,
галузі знань 12 «Інформаційні технології»,
спеціальності 121 «Інженерія програмного забезпечення»,
освітньо-професійної програми «Програмна інженерія»

GAMEHUB: «СПІВРОБІТНИЦТВО МІЖ УНІВЕРСИТЕТАМИ ТА ПІДПРИЄМСТВАМИ В СФЕРІ ІГРОВОЇ ІНДУСТРІЇ В УКРАЇНІ»

GAMEHUB: «UNIVERSITY-ENTERPRISES COOPERATION IN GAME INDUSTRY IN UKRAINE»
561728-EPP-1-2015-1- ES-EPPKA2-CBHE-JP

The handbooks were performed with support of the Erasmus+ Programme of the European Union (561728-EPP-1-2015-1- ES-EPPKA2-CBHE-JP). The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

ХЕРСОН-2018

УДК 004.4
Р 65

Розробка ігрових web додатків: електронний навчальний посібник для підготовки студентів на першому (бакалаврському) рівні вищої освіти, галузі знань 12 «Інформаційні технології», спеціальності 121 «Інженерія програмного забезпечення» / Укладач: М.В. Жарікова. – Херсон: видавництво ФОП Вишемирський В.С., 2018. – 218 с.

Р 65

ISBN 978-617-7573-89-9 (електронне видання)

Укладач:
Жарікова М.В., к.т.н., доцент кафедри Програмних засобів і технологій.

Рецензенти:

Шарко О.В., д.т.н, професор кафедри транспортних технологій Херсонської державної морської академії.

Львов М.С., д.ф-м.н, професор, завідувач кафедри інформатики, програмної інженерії та економічної кібернетики Херсонського державного університету

Розглянуто на засіданні кафедри Програмних засобів і технологій,
Протокол №11 від 23 травня 2017 року.

Рекомендовано до друку Вченою радою Херсмонського національного технічного університету,
Протокол №9 від 2 червня 2017 року.

Цей матеріал ліцензовано на умовах Ліцензії Creative Commons CC BY-NC-SA

Із Зазначенням Авторства — Некомерційна — Поширення На Тих Самих Умовах 4.0 Міжнародна

УДК 004.4

ISBN 978-617-7573-89-9

© Жарікова М.В., 2018
© ФОП Вишемирський В. С., 2018

ЗАГАЛЬНИЙ ВСТУП

Електронний навчальний посібник складений відповідно до програми навчальної дисципліни «Розробка ігрових web-додатків», яка входить до дисциплін підготовки студентів на першому (бакалаврському) рівні вищої освіти, галузі знань 12 «Інформаційні технології», спеціальності 121 «Інженерія програмного забезпечення», освітньо-професійної програми «Програмна інженерія» і містить довідник дисципліни, лекційний матеріал, приклади розв'язку задач з розробки ігрових web-додатків, питання для самоконтролю, задачі для самостійного розв'язку та список рекомендованої літератури.

Основною метою посібника є знайомство студентів з найбільш важливими поняттями і специфічними особливостями створення ігрових web-додатків, а також формування у студентів знань, вмінь та навичок, необхідних для розробки ігрових web-додатків за допомогою мови програмування JavaScript.

Розробником навчального посібника узагальнені й систематизовані теоретичні надбання у галузі розробки ігрових web-додатків.

Запропонована послідовність тем і їх поєднання у три розділи створюють умови для логічного засвоєння змісту дисципліни. Питання, що розкривають сутність відповідних тем, забезпечують викладення методологічних основ дисципліни «Розробка ігрових web-додатків», у разі успішного засвоєння яких студенти будуть вміти:

– застосовувати основні алгоритмічні конструкції мови програмування JavaScript;

– додавати динамічні двовимірні растрові зображення (спрайти) в ігрові веб-додатки;

– додавати звукові ефекти в ігрові веб-додатки;

– реагувати на дії користувача за допомогою подій JavaScript;

– створювати події підтримки роботи з клавіатурою мовою JavaScript в ігрових веб-додатках;

– створювати класи, екземпляри класів та додавати властивості з використанням механізму прототипів мовою JavaScript;

– створювати ігрові об'єкти та підтримувати основні колізії між об'єктами мовою JavaScript.

ЗМІСТ

ДОВІДНИК НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

Вступ		3
1	Опис модуля	3
2	Перелік компетентностей та результати навчання	3
3	Міждисциплінарні зв'язки	6
4	Мета та передбачувані результати вивчення навчального модуля	6
5	Календарний план семестру і структура навчального модуля	7
6	Форми навчання	9
7	Порядок проведення атестації	9
8	Зворотній зв'язок	12
9	Викладацький склад та допоміжні джерела	13
10	Навчальна програма і матеріали	13

ЛЕКЦІЇ ТА МЕТОДИКА ЇХ ПРОВЕДЕННЯ

Вступ		22
1	Лекція 1. Основи побудови ігрових веб-додатків	25
2	Лекція 2. Основні елементи ігор мовою JavaScript	34
3	Лекція 3. Реагування на дії користувача у грі мовою JavaScript	42
4	Лекція 4. Створення обробників подій натискання клавіш у ігрових веб-додатках мовою програмування JavaScript	58
5	Лекція 5. Структурування програмного коду ігрового веб-додатка мовою програмування JavaScript за допомогою об'єктів та методів	67

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Вступ		75
1	Лабораторна робота 1. Створення ігрового світу	75
2	Лабораторна робота 2. Робота з покажчиком миші мовою JavaScript на прикладі гри Painter	90
3	Лабораторна робота 3. Створення ігрового веб-додатка з додаванням динамічних об'єктів	100
4	Лабораторна робота 4. Робота з різними типами об'єктів	110

5	Лабораторна робота 5. Робота з кольором та підтримка колізій між ігровими об ‘єктами	119
6	Лабораторна робота 6. Установка кількості життів в ігрових веб-додатках	129

ПРАКТИЧНІ ЗАНЯТТЯ

1	Практичне заняття 1. Розробка першого веб-додатка	194
2	Практичне заняття 2. Розміщення, завантаження спрайтів та анімація мовою програмування JavaScript	144
3	Практичне заняття 3. Об’єктно-орієнтований підхід до створення ігор мовою JavaScript	153
4	Практичне заняття 4. Контрольований доступ до об’єктів	166

СЕМІНАРИ

1	Семінар 1. Створення ігор з використанням полотна та спрайтів	185
2	Семінар 2. Додаткові можливості використання ігрових циклів	198
3	Семінар 3. Художнє оформлення ігрового веб-додатка мовою JavaScript	201

ЛІТЕРАТУРА 211

Вступ

Метою навчального модуля "Розробка ігрових web-додатків" є знайомство студентів з найбільш важливими поняттями і специфічними особливостями створення ігрових web-додатків, а також формування у студентів знань, вмінь та навичок, необхідних для розробки ігрових web-додатків за допомогою мови програмування JavaScript.

Основними завданнями вивчення навчального модуля "Розробка ігрових web-додатків" є:

- ознайомитись з основами функціонування, налагодження та адміністрування програмного забезпечення, яке реалізує сервіси Інтернет;
- ознайомитись та вивчити теоретичні та аналітичні методи, а також сучасні технологічні підходи до побудови ігрових web-додатків;
- ознайомитись з основами створення ігрових web-додатків за допомогою мови програмування JavaScript, мови розмітки сторінки HTML, а також каскадних таблиць стилів CSS.

Змістовний модуль «Розробка ігрових web-додатків» є частиною орієнтований на оволодіння студентами практичних навичок використання мови програмування JavaScript для створення власних ігрових web-додатків.

1 Опис модуля

Галузь знань: 0501 "Інформатика та обчислювальна техніка".
Напрям підготовки: 6.050103 "Програмна інженерія".
Рівень: бакалавр.
Назва дисципліни: Розробка ігрових web-додатків
Назва змістовного модуля: Розробка ігрових web-додатків

Семестр: **8**
Кількість кредитних одиниць: дисципліна - 3,0, модуль - 3,0
Орієнтовна кількість часів: дисципліна - 240, модуль - 60
Викладач : доцент, к.т.н. Жарікова М.В.

2 Перелік компетентностей та результати навчання

Загальні (універсальні) компетентності

ЗК-1 Здатність до абстрактного мислення, аналізу та синтезу.

ЗК-2 Здатність застосовувати знання в практичних ситуаціях.

ЗК-3 Здатність вчитися й оволодівати сучасними знаннями, здійснювати пошук, оброблення й аналіз інформації з різних джерел.

ЗК-4 Здатність генерувати нові ідеї (креативність), працювати в команді, бути критичним і самокритичним, розробляти проекти, приймати обґрунтовані рішення.

ЗК-5 Здатність оцінювати та забезпечувати якість виконуваних робіт.

ЗК-6 Здатність застосовувати математичний апарат, а також теоретичні, методичні й алгоритмічні основи інформаційних технологій під час вирішення прикладних і наукових завдань в області інформаційних систем і технологій.

Спеціальні (фахові) компетентності

ФК-1 Здатність опановувати сучасні технології математичного моделювання об’єктів, процесів і явищ, розробляти обчислювальні моделі та алгоритми чисельного розв’язання задач математичного моделювання з урахуванням похибок наближеного чисельного розв’язання професійних задач; здійснювати формалізований опис задач дослідження операцій в організаційно-технічних і соціально-економічних системах різного призначення, визначати їх оптимальні рішення, будувати моделі оптимального вибору управління з урахуванням змін параметрів економічної ситуації, оптимізувати процеси управління в системах різного призначення та рівня ієрархії.

ФК-2 Здатність реалізовувати багаторівневі обчислювальні моделі на основі архітектури клієнт-сервер (включаючи сховища, бази та банки даних і знань) для забезпечення обчислювальних потреб багатьох користувачів.

ФК-3 Здатність формулювати та забезпечувати вимоги щодо якості програмного забезпечення у відповідності з вимогами, технічним завданням та стандартами.

ФК-4 Здатність здійснювати аналіз і функціональне моделювання процесів, побудову та застосування функціональних моделей систем; застосування методів та інструментальних засобів для управління процесами життєвого циклу систем відповідно до вимог замовника.

ФК-5 Здатність опановувати та комплексно застосовувати базові загальні знання в області програмування (у тому числі, структурного, функціонального, логічного, об'єктно-орієнтованого, паралельного) та візуального проектування системного та прикладного програмного забезпечення; володіти алгоритмічним мисленням; проектувати та розробляти програмне забезпечення на основі інтеграції провідних сучасних технологій (із застосуванням відповідних моделей, методів та алгоритмів обчислень, структур

даних); застосовувати об'єктно-орієнтований підхід під час проектування складних програмних систем методами програмної інженерії для реалізації програмного забезпечення з урахуванням вимог до його якості, надійності, виробничих характеристик.

ФК-6 Здатність опановувати та комплексно застосовувати базові знання в області принципів, методів і алгоритмів комп'ютерної графіки під час розробки графічних інтерфейсів взаємодії людини з комп'ютером.

ФК-7 Здатність здійснювати процес інтеграції системи, застосовувати стандарти і процедури управління змінами для підтримки цілісності загальної функціональності і надійності програмного забезпечення.

Програмні результати навчання

ПРН-1 Здатність аналізувати проблеми щодо створення програмного забезпечення.

ПРН -2 Здатність, аналізувати, цілеспрямовано шукати і вибирати необхідні для вирішення професійних завдань інформаційно-довідникові ресурси і знання з урахуванням сучасних досягнень науки і техніки.

ПРН -3 Здатність використовувати знання щодо методів та засобів збору, формулювання та аналізу вимог до програмного забезпечення.

ПРН -4 Здатність застосовувати знання ефективних підходів щодо проектування програмного забезпечення.

ПРН -5 Здатність розуміти основні процеси, фази та ітерації життєвого циклу програмного забезпечення.

ПРН -6 Здатність розуміти і застосовувати знання сучасних підходів щодо оцінки та забезпечення якості програмного забезпечення.

ПРН -7 Здатність розуміти і застосовувати знання щодо відповідних математичних понять, методів доменного, системного і об’єктно-орієнтованого аналізів та математичного моделювання для розробки програмного забезпечення.

ПРН-8 Здатність застосовувати на практиці фундаментальні концепції і основні принципи функціонування мовних, інструментальних і обчислювальних засобів інженерії програмного забезпечення

ПРН-9 Здатність застосовувати знання методів компонентної розробки програмного забезпечення, виділяючи інтерфейси і реалізації та взаємодію між модулями, підсистемами і компонентами

Перелік компетентностей та результати навчання
Навчальний модуль «Розробка ігрових web-додатків»

Загальні (універсальні) компетентності	ЗК-4, ЗК-5, ЗК-6
Спеціальні (фахові) компетентності	ФК-1, ФК-2, ФК-3, ФК-5, ФК-6
Програмні результати навчання	ПРН-4, ПРН-7, ПРН-8, ПРН-10

3 Міждисциплінарні зв'язки

Для засвоєння матеріалу використовується такий перелік дисциплін, що забезпечують засвоєння навчального матеріалу дисципліни «Розролка ігрових web-додатків»:

- Вища математика
- Основи програмування
- Основи програмної інженерії
- Алгоритми та структури даних
- Об'єктно-орієнтоване програмування
- Бази даних
- Комп'ютерна графіка
- Розробка мобільних додатків на платформі Android
- Програмування Інтернет

4 Мета та передбачувані результати вивчення навчального модуля

4.1 Мета модуля

Мета модуля – ознайомити студентів з найбільш важливими поняттями і специфічними особливостями мови програмування JavaScript, а також сформувати у студентів знання, вміння та навички, необхідні для розробки комп‘ютерних ігор за допомогою цієї мови

4.2 Результати навчання

Знання та їх використання

У разі успішного оволодіння матеріалами модуля студент буде вміти:

- застосовувати основні алгоритмічні конструкції мови програмування JavaScript;
- додавати динамічні двовимірні растрові зображення (спрайти) в ігрові веб-додатки;
- додавати звукові ефекти в ігрові веб-додатки;
- реагувати на дії користувача за допомогою подій JavaScript;

- створювати події підтримки роботи з клавіатурою мовою JavaScript в ігрових веб-додатках;
- створювати класи, екземпляри класів та додавати властивості з використанням механізму прототипів мовою JavaScript;
- створювати ігрові об'єкти та підтримувати основні колізії між об'єктами мовою JavaScript.

Дослідницькі навички

У разі успішного вивчення модуля студент буде вміти аналізувати завдання щодо створення сценаріїв ігрових web-додатків; обирати найбільш придатну їх архітектуру; оцінювати можливості мови програмування JavaScript для їх реалізації; вміти застосовувати методи анімації об'єктів при розробці ігрового контенту; вміти здійснювати апробацію отриманих результатів, приймаючи участь у відповідних формах організації наукових заходів (семінарах, конференціях, конгресах, симпозіумах тощо); вміти впроваджувати отримані результати у практичну діяльність підприємств та організацій.

Спеціальні вміння

У разі успішного вивчення модуля студент буде вміти:

- Створювати архітектуру ігрового веб-додатка;
- Створювати графічні елементи ігрових веб-додатків мовою програмування JavaScript;
- Створювати крос-платформні ігрові веб-додатки мовою програмування JavaScript.

Соціальні вміння

У разі успішного вивчення модуля студент буде вміти приймати участь в командній роботі: обговорювати в групі архітектуру ігрового веб-додатка, презентувати та аргументувати свої рішення, приймати участь в груповій розробці ігрових додатків мовою програмування JavaScript.

Особисті якості

У разі успішного вивчення модуля студент буде вміти:

- самостійно вирішувати питання, які відносяться до побудови архітектури ігрового веб-додатка мовою програмування JavaScript;
- створювати працездатні ігрові веб-додатки мовою програмування JavaScript та розміщувати їх на хостингу у мережі Інтернет.

5 Календарний план семестру і структура навчального модуля

5.1 Місце модуля в структурі дисципліни

Номер	Змістовний модуль	Тиждень вивчання
1	Розробка ігрових web-додатків для бакалаврів.	1-16
2	Розробка ігрових web-додатків для магістрів.	1-16

5.2 Інформаційне наповнення змістовного модуля 1

Номер тижня	Зміст
1	Основи побудови ігрових веб-додатків. Підходи до побудови архітектур ігрових веб-додатків. Можливості комбінації JavaScript, HTML та CSS для створення простих веб-додатків.
2	Основні елементи JavaScript-гри: ігровий світ і ігровий цикл. Створення простого ігрового додатка за допомогою власних методів та ігрових об'єктів мовою JavaScript. Додавання зображень (спрайтів) у ігрові додатки.
3	Події (event) та обробники подій (event handler) мовою програмування JavaScript. Створення обробників подій для реагування на рух миші.
4	Завантаження та управління рухом динамічних спрайтів. Реагування на дії користувача в ігровому веб-додатку та змінення ігрового світу у відповідності з цими діями мовою програмування JavaScript.
5	Створення обробників подій натискання клавіш у ігрових веб-додатках мовою програмування JavaScript.
6	Структурування програмного коду ігрового веб-додатка мовою програмування JavaScript за допомогою об'єктів та методів. Створення ігрового світу, який містить декілька різних взаємодіючих об'єктів.
7	Створення та робота з класами об'єктів з використанням механізму прототипів мовою програмування JavaScript. Додавання елементу випадковості у ігровий веб-додаток.
8	Додавання властивостей у класи об'єктів. Робота з кольором та підтримка колізій між ігровими об 'єктами.

9	Робота з обмеженою кількістю життів. Перезапуск ігрового веб-додатка за умови, що у гравця не залишилося життів. Робота з циклами.
10-16	Робота с системою GIT. Робота над ігровим веб-додатком у групах. Реєстрація безкоштовного хостингу та розміщення сайту у мережі Інтернет.

6 Форми навчання

Навчальне навантаження модуля складається з аудиторної та самостійної роботи. Аудиторна робота включає 5 лекцій, 6 лабораторних робіт, 4 практичних заняття, 3 семінари, групові або індивідуальні завдання (курсова робота). Самостійна робота студентів передбачає підготовку до аудиторних занять: лабораторних робіт, практичних занять, семінарів, групових та індивідуальних завдань, а також підготовку до заліку та оформлення звітів з лабораторних робіт.

Підготовка до поточних аудиторних занять включає аналіз літератури, Інтернет-матеріалів за темами лекцій, лабораторних робот, практичних занять, семінарів, групових та індивідуальних завдань, підготовку до заліку.

Контактні години передбачають індивідуальні консультації та контроль студентів в он-лайн режимі.

Звіти з лабораторних робіт – опис та презентація UML-діаграм та фрагментів програмного коду ігрових веб-додатків, розроблених мовою програмування JavaScript при виконанні лабораторних робіт 1-5.

7 Порядок проведення атестації

В організації навчального процесу при вивченні дисципліни застосовується поточний та підсумковий контроль. Використовується 100-бальна система оцінювання.

Поточний контроль полягає в:

- оцінюванні навчальних успіхів студентів (слухачів) під час практичних занять (4 заняття по 5 балів = максимально 20 балів);
- оцінюванні участі студентів у семінарах (3 семінари по 5 балів = максимально 15 балів);
- проведенні захисту виконаних студентом (слухачем) лабораторних завдань (6 завдань по 5 балів = максимально 30 балів, якщо лабораторні завдання виконуються та захищаються невчасно, оцінка зменшується на 1 бал за кожне заняття, максимально на 5 балів за кожне заняття);
- проведенні захисту виконаних студентами групових (або індивідуальних) завдань перед студентською аудиторією (максимально 25 балів – по 5 балів за:

(а) ступінь самостійності, завершеності проведеного проектування та відповідності поставленому завданню, (б) глибину проробки, ступінь обґрунтування та адекватності проектних рішень, (в) працездатність та якість оформлення отриманого результату, (г) повноту та якість виконання пояснювальної записки, (д) повноту та якість презентації, в т. ч. ступінь володіння матеріалом, вміння захищати свою думку тощо);

- проведенні модульної контрольної роботи наприкінці семестру (максимально 15 балів). Контрольна робота проводиться у форматі тесту, який містить блок з 5 теоретичних питань, що оцінюються по 1 балу кожне (максимально 5 балів) та 5 практичних завдань, що оцінюються по 2 бали кожне (максимально – 10 балів).

Підсумковий контроль має форму заліку, що заснований на рейтинговому оцінюванні шляхом простого підсумовування всіх результатів поточного контролю навчальних успіхів студента.

Оцінка результатів вивчення дисципліни в цілому

Поточне тестування	**Балів**
Змістовний модуль	до 85
Модульна контрольна робота	до 15
Разом	До 100

Графік проведення поточного оцінювання

Номер тижня	**Оцінювання**
1	Оцінка роботи на практичному занятті 1,
2	Оцінка виконання лабораторної роботи 1
3	Оцінка виконання лабораторної роботи 2 , Оцінка роботи на практичному занятті 2
4	Оцінка роботи на семінарі 1
5	Оцінка виконання лабораторної роботи 3
6	Оцінка виконання лабораторної роботи 4
7	Оцінка роботи на практичному занятті 3, Оцінка роботи на семінарі 2
8	Оцінка виконання лабораторної роботи 5

9	Оцінка роботи на практичному занятті 4, Оцінка роботи на семінарі 3
10-16	Оцінка виконання групових або індивідуальних завдань, Оцінка модульної контрольної роботи

Представлення звіту щодо виконання лабораторних робіт.

При захисті кожної лабораторної роботи студентом надається звіт в електронному вигляді.

За кожний день прострочки представлення та здачі єдиного звіту по модулю 3 знімається 1 бал (не більш ніж 5 днів – 5 балів)

Метод оцінки змістовного модуля

Кількість балів в загальній оцінці змістовного модулю відповідає наступному:

Оцінка роботи на практичному занятті 1,	Максимально 5 бал.
Оцінка виконання лабораторної роботи 1	Максимально 5 бал.
Оцінка виконання лабораторної роботи 2 , Оцінка роботи на практичному занятті 2	Максимально 5 бал. Максимально 5 бал.
Оцінка роботи на семінарі 1	Максимально 5 бал.
Оцінка виконання лабораторної роботи 3	Максимально 5 бал.
Оцінка виконання лабораторної роботи 4	Максимально 5 бал.
Оцінка роботи на практичному занятті 3, Оцінка роботи на семінарі 2	Максимально 5 бал. Максимально 5 бал.
Оцінка виконання лабораторної роботи 5	Максимально 5 бал.
Оцінка роботи на практичному занятті 4, Оцінка роботи на семінарі 3	Максимально 5 бал. Максимально 5 бал.
Оцінка виконання групових або індивідуальних завдань, Оцінка модульної контрольної роботи	Максимально 25 бал. Максимально 15 бал.

Усі набрані бали підсумовуються (максимально 100 балів), штрафні бали за запізнення здачі кожної лобор採торної роботи (максимально мінус 5 балів) віднімаються. Сумарна оцінка (від 0 до 100 балів) є індивідуальна оцінка освоєння студентом змістовного модуля.

Метод оцінки дисципліни в цілому

Підсумковий контроль має форму заліку, що заснований на рейтинговому оцінюванні шляхом простого підсумовування всіх результатів поточного контролю навчальних успіхів студента. Таким чином розраховується сумарна оцінка студента в балах за дисципліною.

Сумарна оцінка в балах, переводиться за нижченаведеною шкалою оцінювання в національну та ЄКТС- оцінку:

Сума балів за всі види навчальної діяльності	Оцінка ECTS	Оцінка за національною шкалою
90 – 100	A	Відмінно
82-89	B	Добре
74-81	C	Добре
64-73	D	Задовільно
60-63	E	Задовільно
35-59	FX	не зараховано з можливістю повторного складання
0-34	F	не зараховано з обов’язковим повторним вивченням дисципліни

8 Зворотній зв’язок

1. Студенти отримують інформацію про результати атестації безпосередньо у викладача.

2. Процедура надання оцінок:

- оцінка лабораторних робіт – на підставі їх прилюдного захисту одразу ж після захисту;
- оцінка навчальних досягнень студента на практичних заняттях та семінарах – прилюдно в кінці заняття;
- оцінка виконання курсових робіт – на підставі їх прилюдного захисту у відповідності до затвердженого графіку;
- оцінка результатів виконання модульної контрольної роботи (на протязі трьох діб після її проведення);
- рейтингова оцінка заліку – наприкінці семестру;
- часові рамки надання поточних та підсумкових оцінок:

захист курсових робіт – 16-й тиждень,

модульна контрольна робота – 16-й тиждень.

3. Консультації проводяться викладачем очно у відповідності до робочого графіку та заочно у режимі електронного листування та електронної конференції на протязі семестру.

Контактні дані для on-line допомоги та консультування:

Викладач : доцент, к.т.н. Жарікова М.В., marina.jarikova@gmail.com

9 Викладацький склад та допоміжні джерела

Обов'язки викладачів

Основні обов'язки викладачів навчальної дисципліни полягають у проведенні лекцій і лабораторних занять згідно навчальної програми та проведенні контролю якості отриманих знань, умінь і навичок.

Обов'язки координатора дисципліни

Головні обов’язки координаторів навчальної дисципліни полягають у розробці та внесенні змін до змістовних модулів у відповідності з поточними потребами, навчальними планами тощо; координації і управлінні професорсько-викладацьким складом; координації проведення заліків.

Обов'язки допоміжного персоналу

Допоміжний персонал здійснює підготовку комп’ютерної техніки та спеціалізованого ігрового обладнання до виконання лабораторних робіт студентами та надає технічну підтримку студентам під час виконання лабораторних робіт.

10 Навчальна програма і матеріали

10.1 Тема 1: Основи побудови ігрових веб-додатків. Підходи до побудови архітектур ігрових веб-додатків. Можливості комбінації JavaScript, HTML та CSS для створення простих веб-додатків

10.1.1 Мета та очікувані результати

Ознайомити студентів з основними підходами до побудови архітектур ігрових веб-додатків. Узагальнити матеріал щодо можливостей комбінації JavaScript, HTML та CSS для створення простих веб-додатків.

Перевірити рівень знань мови розмітки сторінки HTML та каскадної таблиці стилів CSS.

10.1.2 Лекція 1

Лекція знайомить з основами побудови ігрових веб-додатків мовою програмування JavaScript. Розглядаются підходи до побудови архітектур ігрових веб-додатків. Виконується тестова перевірка рівня знань мови розмітки сторінки HTML та каскадних таблиць стилів CSS.

Мета лекції:

- Перевірити рівень знань мови розмітки сторінки HTML та каскадних таблиць стилів CSS;
- Ознайомити студентів з базовими підходами до побудови архітектур ігрових веб-додатків;
- Ознайомити студентів з основами побудови ігрових веб-додатків мовою програмування JavaScript.

Основні результати лекції відповідають вище передбаченим цілям.

10.1.3 Практичне заняття 1. Розробка першого веб-додатка

Практичне заняття орієнтовано на ознайомлення студентів з побудовою простого ігрового веб-додатка с використанням тега HTML canvas та елементів мови JavaScript.

Мета практичного заняття:

- Відновити в пам'яті основні елементи структури HTML-документу.
- Створити простий веб-додаток з додаванням HTML-тега canvas, який дозволяє додавати графічні елементи в документ.
- Навчитися виносити JavaScript-код в окремий файл.

У разі успішного проходження практичного заняття студент буде знати структуру HTML-документу, способи додавання JavaScript-кода в код HTML, вміти створювати простий веб-додаток з додаванням HTML-тега canvas, який дозволяє додавати графічні елементи в документ.

10.1.4 Лабораторна робота 1. Створення ігрового світу

Лабораторна робота орієнтована на ознайомлення студентів з основами побудови ігрового світу мовою програмування JavaScript.

Мета лабораторної роботи:

- Ознайомити студентів з основними типами змінних, а також об'єктами, які містять змінні-члени та методи.
- Ознайомити студентів з декларацією та ініціалізацією змінних, з поняттям глобальні змінні, операторами мови JavaScript.
- Навчити студентів створювати простий веб-додаток, в якому геометрична фігура переміщується по полотну.

У разі успішного виконання лабораторної роботи студент буде знати, як зберігати інформацію в змінних, як створювати об'єкти, які містять змінні-члени та методи, як використовувати метод update для зміни ігрового світу та метод draw для відображення ігрового світу на екрані.

10.1.5 Питання, що виносяться на іспит.

- Визначте структуру HTML-документа.
- Поясніть, як JavaScript-код винести в окремий файл.
- Визначте поняття ігрового світу.
- Визначте поняття ігрового циклу.
- Визначити основні типи змінних, поняття ініціалізації змінних, глобальних змінних, основні оператори мови програмування JavaScript.

10.2 Тема 2. Основні елементи ігор мовою JavaScript.

10.2.1 Мета та очікувані результати

Навчити студентів додавати зображення (спрайти) в ігрові додатки. Ознайомити студентів з поняттям подій (event) та обробників подій (event handler) мовою програмування JavaScript.

Сформувати у студентів вміння, навички та основні прийоми створення обробників подій для реагування на рух миші.

10.2.2 Лекція 2

Лекція знайомить студентів з основними елементами ігор мовою JavaScript, зі структурою ігрової програми, макетом.

Мета лекції:

- Навчити студентів структурувати ігрову програму.
- Навчити студентів дотримуватисьправил макетування програм, розміщення коментарів.

Основні результати лекції відповідають вище передбаченим цілям.

10.2.3 Практичне заняття 2. Розміщення, завантаження спрайтів та анімація мовою програмування JavaScript

Практичне заняття орієнтовано на отримання студентами практичних навичок розміщення, завантаження спрайтів, та створення динамічних спрайтів мовою JavaScript.

Мета практичного заняття:

- Навчити студентів завантажувати спрайти з файлів.
- Навчити студентів розміщувати спрайти на екрані.
- Навчити студентів створювати ігровий цикл для переміщення спрайтів.

У разі успішного проходження практичного заняття студент буде знати основні прийоми роботи зі спрайтами мовою програмування JavaScript.

10.2.4 Лабораторна робота №2. Робота з покажчиком миші мовою JavaScript на прикладі гри Painter.

Лабораторна робота орієнтована на оволодіння студентами навичок роботи зі спрайтами мовою програмування JavaScript.

Мета лабораторної роботи:

- Оволодіти прийомами додавання декількох спрайтів в ігровий веб-додаток.
- Освоїти навички сворення ігрового циклу для переміщення спрайтів.
- Освоїти навички створення методу update для розрахунку позиції для відображення спрайту, та методу draw для відображення.
- Навчитися додавати звукові ефекти в ігрові веб-додатки.

У разі успішного виконання лабораторної роботи студент буде вміти додавати динамічні спрайти в ігровий веб-додаток.

10.2.5 Питання, що виносяться на іспит.

- Як отримати інформацію з файла зображення мовою JavaScript?
- Як розмістити спрайт на екрані?
- Як створити динамічний спрайт?
- Як додати звукові ефекти в ігровий додаток?

10.3 Тема 3: Реагування на дії користувача у грі мовою JavaScript.

10.3.1 Мета та очікувані результати

Ознайомити студентів з прийомами завантаження та управління рухом динамічних спрайтів. Навчити студентів реагувати на дії користувача в ігровому веб-додатку та змінювати ігровий світ у відповідності з цими діями мовою програмування JavaScript

10.3.2. Лекція 3

Лекція знайомить студентів з організацією руху спрайту, який повторює рух покажчика миші. Пояснюється створення обробника події руху миші. Пояснюється динаміка спрайту як реакція на подію руху миші.

Мета лекції:

- Ознайомити студентів з базовим прийомом отримання позиції покажчика миші.
- Сформувати у студентів навички створення динамічних спрайтів, які повторюють рух покажчика миші.

Основні результати лекції відповідають вище передбаченим цілям.

10.3.2 Семінар №1. Створення клієнтських ігрових веб-додатків мовою програмування JavaScript.

Семінар орієнтований на отримання студентами знань про клієнт-серверну архітектуру веб-додатків та особливості мови JavaScript як серверної мови програмування.

Мета семінару:

- Освоїти клієнт-серверну архітектуру ігрових веб-додатків.

- Вивичити особливості мови JavaScript як серверної мови програмування.
- Розглянути особливості розробки серверних ігрових веб-додатків мовою програмування JavaScript.

У разі проходження семінару студент буде знати особливості клієнт-серверної архітектури ігрових веб-додатків, особливості мови JavaScript як серверної мови програмування та особливості розробки серверних ігрових веб-додатків мовою програмування JavaScript.

10.3.3 Питання, що виносяться на іспит.

- Поясніть, як отримати позицію покажчика миші мовою програмування JavaScript.
- Поясніть, як створити динамічний спрайт, який повторює рух покажчика миші.

10.4 Тема 4: Створення обробників подій натискання клавіш у ігрових веб-додатках мовою програмування JavaScript.

10.4.1 Мета та очікувані результати

Ознайомити студентів з основами створення обробників подій натискання клавіш у ігрових веб-додатках мовою програмування JavaScript.

10.4.2 Лекція 4

Лекція знайомить з прийомами реагування на натискання клавіш.

Розглядаються додаткові можливості структурування коду за допомогою об'єктів та методів.

Мета лекції:

- Ознайомити студентів з прийомами реагування на натискання клавіш.
- Ознайомити студентів зі способом підтримки події натискання клавіші.
- Довести до студентів додаткові можливості структурування коду за допомогою об'єктів та методів.

Основні результати лекції відповідають вище передбаченим цілям.

10.4.3 Лабораторна робота №3. Створення ігрового веб-додатка з підтрмкою події натискання клавіші мовою JavaScript:

Лабораторна робота орієнтована на оволодіння студентами порядком та послідовністю дій щодо створення закінченого ігрового додатку з підтримкою реагування на натискання клавіш мовою JavaScript.

Мета лабораторної роботи:

- Освоїти прийоми підтримки події натискання клавіші.
- Створити ігровий веб-додаток з підтримкою реагування на натискання клавіш мовою JavaScript.

У разі успішного виконання лабораторної роботи студент буде вміти створювати ігровий веб-додаток з підтримкою реагування на натискання клавіш мовою JavaScript.

10.4.4 Питання, що виносяться на іспит.

- Поясніть механізм обробки події натискання клавіші мовою програмування JavaScript.
- Оператори порівняння, логічні оператори мови JavaScript.

10.5 Тема 5: Структурування програмного коду ігрового веб-додатка мовою програмування JavaScript за допомогою об'єктів та методів. Створення ігрового світу, який містить декілька різних взаємодіючих об'єктів. Створення та робота з класами об'єктів з використанням механізму прототипів мовою програмування JavaScript. Додавання елементу випадковості у ігровий веб-додаток.

10.5.1 Мета та очікувані результати

Ознайомити студентів зі способом структурування програмного коду ігрового веб-додатка мовою програмування JavaScript. Навчити створювати ігровий світ, який містить декілька різних взаємодіючих об'єктів, та додавати елемент випадковості у ігровий веб-додаток.

10.5.2 Лекція 5

Лекція знайомить з прийомами структурування програмного кода з використанням об'єктів та за допомогою розбиття кода на окремі файли.

Мета лекції:

- Ознайомити студентів з прийомами опису об'єктів в окремих файлах.
- Ознайомити студентів зі способом відокремлення загального коду та ігрового коду.

Основні результати лекції відповідають вище передбаченим цілям.

10.5.3 Лабораторна робота №4. Створення структурованого ігрового веб-додатка мовою JavaScript.

Лабораторна робота орієнтована на оволодіння студентами навичками створення структурованого JavaScript-кода.

Мета лабораторної роботи:

- Навчити студентів прийомам опису об'єктів в окремих файлах.
- Навчити студентів відокремлювати загальний код від ігрового коду.
- Навчити студентів створювати ігровий веб-додаток, в якому загальний код буде відокремлений від ігрового.

У разі успішного виконання лабораторної роботи студент буде вміти створювати ігровий веб-додаток, в якому загальний код буде відокремлений від ігрового, мовою JavaScript.

10.5.4 Практичне заняття 3. Типи ігрових об'єктів.

Практичне заняття орієнтовано на отримання студентами практичних навичок створення та управління об'єктами мовою JavaScript.

Мета практичного заняття:

- Надати студентам поняття концепції класу.
- Навчити студентів додавати елемент випадковості в ігрові веб-додатки.

У разі успішного проходження практичного заняття студент буде знати основні прийоми роботи з класами та додавати елемент випадковості в ігрові веб-додатки мовою програмування JavaScript.

10.5.5 Семінар №2. Створення ігрових об'єктів як частини ігрового світу.

Семінар орієнтований на отримання студентами знань про створення класів об'єктів.

Мета семінару:

- Освоїти прийоми створення класів об'єктів з використанням механізму прототипів.
- Вивчити особоливості створення ігрових об'єктів.
- Освоїти прийоми додавання елементів випадковості в ігрові веб-додатки.

У разі проходження семінару студент ознайомиться з особливостями створення класів об'єктів з використанням механізму прототипів, а також з прийомами додавання елементів випадковості в ігрові веб-додатки мовою програмування JavaScript

10.5.6 Питання, що виносяться на іспит.

- Як розподілити код між різними файлами?
- Як створити клас об'єктів?
- Як додати елемент випадковості в ігровий веб-додаток?

10.6 Тема 6: Додавання властивостей у класи об'єктів. Робота з кольором та підтримка колізій між ігровими об'єктами. Робота з обмеженою кількістю життів. Перезапуск ігрового веб-додатка за умови, що у гравця не залишилося життів. Робота з циклами.

10.6.1 Мета та очікувані результати

Навчити студентів додавати властивості до класів об'єктів. Ознайомити студентів зі способом роботи з кольором та підтримкою колізій між ігровими об 'єктами, з роботою з обмеженою кількістю життів.

10.6.2 Лабораторна робота №5. Робота з кольором та підтримка колізій між ігровими об 'єктами.

Лабораторна робота орієнтована на оволодіння студентами навичками створення властивостей об'єктів, прийомами роботи з кольором.

Мета лабораторної роботи:

- Навчити студентів прийомам створення властивостей об'єктів.
- Навчити студентів прийомами роботи з кольором.

У разі успішного виконання лабораторної роботи студент буде володіти прийомами створення властивостей об'єктів та роботи з кольором.

10.6.3 Лабораторна робота №6. Контрольований доступ до об'єктів.

Лабораторна робота орієнтована на оволодіння студентами навичками створення модифікаторів доступу до властивостей об'єктів.

Мета лабораторної роботи:

- Навчити студентів прийомам створення модифікаторів доступу до властивостей об'єктів.
- Навчити студентів підтримувати колізії між ігровими об'єктами.

У разі успішного виконання лабораторної роботи студент буде вміти створювати модифікатори доступу до властивостей об'єктів мовою JavaScript.

10.6.4 Практичне заняття 4. Установка кількості життів в ігрових веб-додатках.

Практичне заняття орієнтовано на отримання студентами практичних навичок роботи з обмеженою кількістю життів в ігрових веб-додатках мовою JavaScript.

Мета практичного заняття:

- Навчити студентів додавати обмежену кількість життів гравця.
- Навчити студентів обробляти цикли мовою JavaScript.

У разі успішного проходження практичного заняття студент буде знати основні прийоми роботи з обмеженою кількістю життів в ігрових веб-додатках мовою програмування JavaScript.

10.6.5 Семінар №3. Робота з обмеженою кількістю життів. Перезапуск ігрового веб-додатка за умови, що у гравця не залишилося життів. Робота з циклами.

Семінар орієнтований на отримання студентами навичок роботи з обмеженою кількістю життів гравця.

Мета семінару:

- Освоїти прийоми роботи з обмеженою кількістю життів.
- Навчитися перезапускати ігровий веб-додаток за умови, що у гравця не залишилося життів.

У разі успішного проходження практичного заняття студент буде знати основні прийоми роботи з обмеженою кількістю життів в ігрових веб-додатках мовою програмування JavaScript.

10.6.6 Питання, що виносяться на іспит.

- Як створити властивості об'єктів?
- Як створити клас об'єктів?
- Як створити модифікатори доступу до властивостей об'єктів?

10.7 Рекомендована література (інтернет посилання)

1. Ehhes A. Building JavaScript games for phones, tablets, and desktop. – Apress, 2014. – 410 p.
2. Spuy R. Advanced game design with HTML5 and JavaScript. – Apress, 2015. – 526 p.
3. Bunyan K. Build an YTML game: a developer's guide with CSS and JavaScript. – San Francisco. – 220 p.
4. Brown E. Learning JavaScript. – O'Reilly Media. – 364 p.
5. Бахирев А.М. Сюрреализм на JavaScript. – Санкт-Петербуерг: СОНЭЛ, 2014. – 228 с.
6. Ehhes A. Building JavaScript games for phones, tablets, and desktop. – Apress, 2014. – 410 p.
7. Spuy R. Advanced game design with HTML5 and JavaScript. – Apress, 2015. – 526 p.
8. Bunyan K. Build an YTML game: a developer's guide with CSS and JavaScript. – San Francisco. – 220 p.
9. Brown E. Learning JavaScript. – O'Reilly Media. – 364 p.
10. Бахирев А.М. Сюрреализм на JavaScript. – Санкт-Петербуерг: СОНЭЛ, 2014. – 228 с.

ЛЕКЦІЇ ТА МЕТОДИКА ЇХ ПРОВЕДЕННЯ

Вступ

Лекції є основною формою проведення навчальних занять, що призначені для засвоєння теоретичного матеріалу.

При проведенні лекцій з дисципліни «Програмування ігрових web-додатків» лектор повинен дотримуватись таких вимог:

1. Доведення до студентів мети лекції та належне її мотивування. Це виховує в них уміння одразу, без зволікань, залучатися до процесу слухання лекції.

2. Доступність і науковість викладу. Доступність передбачає врахування рівня студентів, їх індивідуальних особливостей, а науковість – розкриття причинно-наслідкових зв'язків, явищ, подій, проникнення в їх сутність, міждисциплінарні зв'язки тощо. Матеріал має бути цікаво вибудуваний, щоб легко сприймався і повніше й всебічніше усвідомлювався студентом. Викладач має відстежувати, що зі сказаного ним і якою мірою сприйнято аудиторією, чи не виникли у слухачів запитання через недостатнє розуміння змісту лекції, непідготовленість до її сприйняття; чи встигають вони усвідомити кожне нове положення, чи вміють поєднувати нову інформацію з попередньою тощо.

3. Включення механізму зворотного зв'язку. Це дає змогу лектору не лише контролювати рівень сприймання, а й регулювати процес роздумів залежно від реального стану студентів.

4. Повторення важливих теоретичних положень. Такі повтори підвищують імовірність запам'ятовування, а отже, і розуміння, систематизацію матеріалу, який ґрунтується на міцному фундаменті засвоєних фактів.

5. Завершення кожного питання лекції підсумком і мотивованим переходом до наступного.

6. Емоційність викладу. Вона є засобом мобілізації і підтримання уваги студентів. Емоційність досягається насамперед чіткою, живою, образною, інтонованою мовою викладача. їй сприятимуть також афоризми, вдалі аналогії, ідіоматичні вирази.

7. Налагодження живого контакту. Йдеться про вміння викладача тримати в полі свого зору кожного студента, своєчасно і правильно реагувати на їх міміку, репліки, жести, вдало використати жарт, дотеп, гумор. Такі засоби зближують викладача з аудиторією і сприяють створенню настрою для більш осмисленого сприймання змісту лекції.

8. Створення проблемних ситуацій. Усвідомлення студентами проблеми налаштовує їх на її розв'язання, спонукає до роздумів, активізує їх пізнавальну діяльність.

Рівень та послідовність викладення лекційного матеріалу відповідає освітньо–професійній програмі підготовки фахівців на першому (бакалаврському) рівні вищої освіти галузі знань 12 «Інформаційні технології», спеціальності 121 «Інженерія програмного забезпечення» за спеціалізацією

«Розробка комп'ютерних ігор».

Зміст лекційного матеріалу характеризується достатньою простотою, що дозволяє сподіватися на його глибоке засвоєння студентами.

Лекція 1
Основи побудови ігрових веб-додатків.

Анотація. Лекція знайомить з історією виникнення веб-додатків та ігрових веб-додатків, основами побудови ігрових веб-додатків мовою програмування JavaScript. Розглядаются підходи до побудови архітектур ігрових веб-додатків. Виконується тестова перевірка рівня знань мови розмітки сторінки HTML та каскадних таблиць стилів CSS.

Мета лекції:

- Перевірити рівень знань мови розмітки сторінки HTML та каскадних таблиць стилів CSS;
- Ознайомити студентів з базовими підходами до побудови архітектур ігрових веб-додатків;
- Ознайомити студентів з основами побудови ігрових веб-додатків мовою програмування JavaScript.

1.1. Програмування

У цій лекції розповідається про те, як мови програмування змінювалися з часом. З моменту виникнення Інтернету в 1990-х роках для його підтримки було розроблено багато мов та інструментів. Однією з найвідоміших мов є HTML, яка використовується для створення веб-сайтів. Разом з таблицями стилів JavaScript та CSS це дозволяє створювати динамічні веб-сайти, які можуть відображатися за допомогою браузера

Комп'ютери та Програми

Перш ніж почати працювати з HTML та JavaScript, цей розділ коротко розглядає комп'ютери та програмування взагалі. Після цього ви перейдете до того, як створити просту HTML-сторінку в поєднанні з JavaScript.

Процесор і Пам'ять

Загалом, комп'ютер складається з *процесора* та *пам'яті* .Це вірно для всіх сучасних комп'ютерів, включаючи ігрові консолі, смартфони та планшети. Я визначаю пам'ять як те, що ви можете *читати речі, і / або написати речі* .Пам'ять поставляється в різних різновидах, в основному відрізняється швидкістю передачі даних та доступу до даних. Деяку пам'ять можна читати та записувати стільки разів, скільки хочеш, деяку пам'ять можна читати, а іншу пам'ять можна записати тільки до

Основний процесор на комп'ютері називається *центральним процесором (ЦП)*. Найпоширенішим іншим процесором на комп'ютері є *графічний процесор (GPU)*. Навіть сам процесор в даний час більше не є єдиним процесором, але

часто складається з декількох ядер. Вхідні та вихідні пристрої, такі як миша, геймпад, клавіатура, монітор, принтер, сенсорний екран тощо, на перший погляд, виходять за межі *процесора* та категорій *пам'яті*. Однак абстрактно кажучи, вони насправді пам'ять. Сенсорний екран *тільки* для *читання* пам'яті, а принтер *тільки* для *запису* пам'яті.

Основним завданням процесора є виконання *інструкцій* .Ефект від виконання цих інструкцій полягає в тому, що пам'ять змінюється. Особливо з моїм дуже широким визначенням пам'яті, кожна команда, яку процесор виконує, певним чином змінює пам'ять .Ви, напевно, не хочете комп'ютера до виконувати тільки один інструкція Як правило, ви мати а дуже довго список від вказівки до буде виконано- "Перемістіть цю частину пам'яті туди, очистіть цю частину пам'яті, малюньте цей спрайт на екрані, перевірте, чи плеєр натискає клавішу на геймпад, і робить каву, коли ви на ньому" - і (як ви ймовірно очікуєте) такий список інструкцій, який виконується комп'ютером, називається *програмою* .

Програми

Таким чином, програмою є довгий список інструкцій щодо зміни пам'яті комп'ютера. Однак сама програма також зберігається в пам'яті. Перед виконанням інструкцій у програмі вони зберігаються на жорсткому диску, DVD або флеш-диску USB; або в хмарі; або на будь-якому іншому носії для зберігання. Коли вони потребують виконання, програма переноситься у внутрішню пам'ять машини.

Інструкції, які об'єднуються разом, формують програму, повинні бути певним чином виражені.

Комп'ютер не може сприймати інструкції, введені простою англійською мовою, тому вам потрібні мови програмування, такі як JavaScript. На практиці інструкції кодуються як текст, але ви повинні слідувати дуже строгим способом їх складання, відповідно до набору правил, які визначають мову програмування. Багато мов програмування існують, тому що коли хтось думає про дещо кращий спосіб вираження певного типу інструкцій, їхній підхід часто стає новою мовою програмування. Важко сказати , скільки мов програмування є, тому що залежить про те, чи ви вважаєте всі версії та діалекти мови; але достатньо сказати, що є тисячі.

На щастя, не потрібно вивчати всі ці різні мови, оскільки вони мають багато спільних рис. Основна мета мов програмування в перші дні - використання нових можливостей комп'ютерів. Однак останнім часом мови зосереджуються на тому, щоб привести певний порядок до хаосу, який може спричинити написання програм. Мови програмування, що мають подібні властивості, належать до тієї ж *парадигми програмування* .Парадигма означає набір практик, який широко використовується.

Ранні дні: імперативне програмування

Велика група мов програмування належить до *імперативної парадигми* .Тому ці мови називаються *імперативними мовами* .Необхідні мови засновані

на інструкціях щодо зміни пам'яті комп'ютера. Таким чином, вони добре підходять для моделі процесор-пам'яті, описаної в попередньому розділі. JavaScript є прикладом імперативної мови.

Перші дні, програмування комп'ютерних ігор було дуже важким завданням, яке вимагало великої майстерності. Ігрова консоль, як і популярна Atari VCS, мала всього 128 байт оперативної пам'яті (Random Access Пам'ять)

і міг використовувати картриджі з максимально 4096 байтами ROM (Memory-Read Only), які повинні містити як програму, так і дані гри. Це значно обмежило можливості. Наприклад, більшість ігор мали симетричний дизайн рівня, оскільки це зменшило вимоги до пам'яті наполовину. Машини також були надзвичайно повільні.

Програмування таких ігор проводилося на мові *асемблерів* .Мова асемблерів - це перші імперативні мови програмування. Кожен тип процесора мав власний набір асемблерних інструкцій, тому ці асемблерні мови були різними для кожного процесора. Тому що таке

доступна обмежена кількість пам'яті, програмісти-грати були експертами з витирання останніх бітів пам'яті та виконання надзвичайно розумних хаків для підвищення ефективності. Однак остаточні програми були нечитаними і не могли бути зрозумілі нікому, крім оригінального програміста. На щастя, це не було проблемою, адже тоді ігри були розроблені окремо людина

Велика проблема полягала в тому, що кожен процесор мав власну версію асемблерської мови, кожен раз, коли з'явився новий процесор, всі існуючі програми повинні були бути повністю переписані для цього процесора. Тому виникла потреба у незалежних від процесорів мов програмування.

Це призвело до таких мов, як *Fortran* (FORmula TRANslator) та універсальний символічний код інструкції *BASIC* (для початківців).BASIC був дуже популярним у 1970-х роках, тому що він прийшов з ранніми персональними комп'ютерами, такими як Apple II в 1978 році, IBM-PC в 1979 році, та їхні нащадки. На жаль, ця мова ніколи не була стандартизованою, тому кожен комп'ютерний бренд використовував власний діалект BASIC.

Процесуальне програмування: імперативний + процедури

Оскільки програми стали більш складними, було зрозуміло, що краще було б організувати всі ці інструкції. У *парадигмі процедурного програмування* пов'язані інструкції згруповані разом у *процедурах* (або *функціях* або *методах* , останньою з яких є найпоширеніша сучасна назва).Оскільки процедурна мова програмування як і раніше містить інструкції, всі процедурні мови також є обов'язковими.

Однією з відомих процедурних мов є C. Ця мова була визначена Bell Labs, яка працювала над розробкою в операційній системі Unix наприкінці 1970-х років. Оскільки операційна система є дуже складною програмою, Bell Labs хотів написати це на процедурній мові. Компанія визначила нову мову, що називається C (тому що вона була наступником попередніх прототипів A та B). Філософія Unix полягала в тому, що кожен може написати свої власні розширення для операційної системи, і було сенс написати ці розширення в C.

Як результат, C став найважливішою процедурною мовою 1980-х років, також поза межами Unix світ C все ще використовується досить багато, хоча це повільно, але безсумнівно, дозволяє створювати нові сучасні мови, особливо в ігровій індустрії.Протягом багатьох років ігри стали набагато більшими програмами, і вони були створені командами, а не окремими особами. Важливо, щоб код гри був читабельним, багаторазовим і простим для налагодження. Також, з фінансової точки зору, все більш важливим стає скорочення часу, коли програмісти повинні були працювати над грою. Хоча C було набагато краще в цьому відношенні, ніж асемблерські мови, залишалося важко писати дуже великі програми в структурованому вигляді шлях

Об'єктно-орієнтоване програмування: процедурний + об'єкти

Процедурні мови, такі як C, дозволяють групувати інструкції в процедурах (також називаються *методами*). Так само, як вони зрозуміли, що інструкції належать групам, програмісти бачили, що деякі методи належать також. *Об'єктно-орієнтована парадигма* дозволяє методу програмісти групи в той , що називається *класом. Пам'ять, яку ці групи методів може змінювати, називається об'єктом* .Клас може описати щось на зразок привидів у грі Pac-Man. Тоді кожен індивідуальний привид відповідає об'єкту класу. Цей спосіб мислення про програмуванні є потужним при застосуванні до ігри

Всі вже програмували в C, тому була закладена нова мова, яка нагадувала C, але це дозволило програмістам використовувати класи та об'єкти. Ця мова називалася *C* ++ (два знаки плюс показали, що він був наступником C).Перша версія C ++ датована в 1978 році, і з'явився офіційний стандарт 1981 рр.

Хоча мова C ++ є стандартною, C ++ не містить стандартний спосіб писати програми на базі Windows на різних типах операційних систем. Написання такої програми на комп'ютер Apple, комп'ютер з ОС Windows або комп'ютер Unix - це зовсім інше завдання, що робить програми C + + різними операційними системами складною проблемою. Спочатку це не вважалося проблемою; але оскільки Інтернет став більш популярним, все більше зручно користуватись можливістю запуску однієї і тієї ж програми в різних операційних системах.

Настав час для нової мови програмування: така, яка буде стандартизована для використання в різних операційних системах. Мова повинна бути схожою на C ++, але це також була чудова можливість видалити деякі старі C-файли з мови, щоб спростити речі. The мова *Java* виконував цю роль (Java є індонезійським островом, що славиться своєю кавою).Java була запущена в 1995 році апаратним виробником Sun, який за цей час використовував революційну бізнес-модель: програмне забезпечення було безкоштовним, і компанія планувала заробляти гроші за допомогою підтримки. Важливим для Sun також було необхідність конкурувати з зростаючою популярністю програмного забезпечення Microsoft, яке не працювало на комп'ютерах Unix, вироблених Сонце

Однією з новинок Java була те, що мова була розроблена таким чином, щоб програми не могли випадково втручатися в інші програми, що працюють на одному комп'ютері. У C ++ це стає значною проблемою: якщо така помилка

сталася, це може призвести до збою всього комп'ютера, або ж гірше - погані програмісти можуть вводити віруси та шпигунські програми.

Веб-додатки

Одним з цікавих аспектів Java було те, що її можна запустити в браузері як так званий *аплет* .Це дало можливість спільного використання програм через Інтернет. Проте запуск аплету Java вимагає встановлення плагіна; і, крім того, немає простої можливості для Java-аплета взаємодіяти з елементами браузера. Звичайно, інше головне завдання браузера - показати *HTML-сторінки* .HTML - мова форматування документа, це абревіатура *мови розмітки HyperText*. Його мета - забезпечити спосіб структурування документів відповідно до набору тегів, які вказують різні частини документа, наприклад, заголовок або абзац. HTML був винайдений наприкінці 1980-х років фізик Тім Бернерс-Лі, який в той час працював у CERN у Швейцарії. Він хотів запропонувати дослідникам CERN легко використовувати та обмінюватися документами. Отже, в записці до своїх колег-дослідників він запропонував Інтернет-гіпертекстову систему. Berners-Lee вказав невеликий набір тегів, які програма перегляду HTML може розпізнати. У першій версії HTML міститься 18 з цих тегів, а 11 з них все ще в сучасному HTML.

Коли Інтернет стає загальнодоступним, HTML став загальноприйнятою мовою для створення веб-сайтів у всьому світі. Тоді дуже популярний браузер Mosaic представив новий тег img , який можна було б використати для включення зображення в документ HTML.Крім того, ряд нових версій HTML-мови розроблявся різними групами, які пропонували стандартизувати певні елементи, які вже були реалізовані багатьма браузерами, такими як таблиці або заповнення форми

У 1995 році стандарт HTML 2.0 був розроблений робочою групою HTML, яка включала всі ці елементи в єдиний стандарт. Після цього консорціум World Wide Web (W3C) був створений для підтримки і оновлення стандарту HTML з плином часу. Нова версія HTML, HTML 3.2 була визначена в січні 1997 року. У грудні того ж року W3C рекомендував HTML4; і, нарешті, HTML4.01 став нещодавно прийнятим стандартом у травні 2000 року.

На всякий випадок, якщо ви ніколи не створювали веб-сайт, так виглядає проста HTML-сторінка:

```
<0,100DOCTYPE html>
<html>
<head>
<title> Корисний веб-сайт </ title>
</ head>
<body>
Це дуже корисний веб-сайт.
</ body>
</ html>
```

Компанії, які розробляли веб-переглядачі, незабаром зрозуміли, що їм потрібен спосіб зробити сторінки динамічнішими. Перший HTML стандарт (2.0) був дуже спрямований на розмітку тексту (саме тому HTML був винайдений в першу чергу). Однак користувачам веб-сайту потрібні кнопки та поля, і потрібна була специфікація, яка б вказувала, що повинно відбутися, якщо користувач взаємодіє з сторінкою.Іншими словами, веб-сайти повинні стати більш динамічними. Звичайно, там була Java з її аплетами, але ці апплети побігли повністю незалежно. Для аплету не можна було змінювати елементи HTML сторінка

Компанія Netscape, компанія, яка розробляла браузер Netscape Navigator, перебувала в жорсткій конкуренції з Microsoft, через яку браузер стане основним, яким усі користуються. У деяких існуючих інструментах Netscape використовувала мову програмування Java, і компанія хотіла спроектувати легку інтерпретовану мову, яка могла б звернутися до непрофесійних програмістів (таких як веб-дизайнери). Ця мова зможе взаємодіяти з веб-сторінкою і читати або змінювати його вміст динамічно. Netscape винайшов мову під назвою *LiveScript* для виконання цієї ролі.Небагато пізніше компанія змінила назву мови сценарію на *JavaScript* , враховуючи її коріння в мові Java і, ймовірно, тому, що люди вже визнали назву Java.JavaScript включено в Netscape Navigator 2.0 JavaScript швидко отримав широкомасштабний успіх як мова скриптів, що дозволило веб-сайтам стати більш динамічними.Корпорація Майкрософт також включила її в Internet Explorer 3.0, але називала її *JScript,* оскільки вона була дещо іншою версією, відокремленою від Netscape.У 1996 році Netscape представив JavaScript в організацію стандартизації ECMA, яка перейменована на мову ECMAScript (хоча все ще називає це JavaScript). Версія, яка нарешті була прийнята у 1999 році як стандарт, є версією, яку підтримують всі поточні браузери. Остання версія стандарту ECMAScript - версія 5.1, випущена в 2011 році. ECMAScript 6, який знаходиться в розробці, представляє багато корисних нових функцій, таких як класи та значення за замовчуванням для функції параметри.

Завдяки підтримці всіх основних веб-переглядачів, JavaScript став основною мовою програмування для веб-сайтів. Оскільки вона спочатку була задумана як легка, інтерпретована мова скриптів, лише зараз програмісти починають використовувати JavaScript для розробки більш складних веб-додатків.

Незважаючи на те, що у JavaScript, можливо, не є всі можливості сучасних мов програмування, таких як Python та C #, це все ще дуже сильна мова, як ви відкриєте під час читання цієї книги. В даний час JavaScript є єдиною мовою, інтегрованою з HTML, яка працює в різних браузерах на різних платформах. Разом з HTML5 він став потужною основою для веб-розробки.

Програмування ігор

Мета модуля, навчити, як програмувати ігри. Ігри дуже цікаві (а іноді й складні!) . Вони мають справу з безліччю різних пристроїв вводу та виводу, і уявні світи, створені іграми, можуть бути надзвичайно складними.

До початку 90-х років ігри були розроблені для конкретних платформ. Наприклад, гра, написана для конкретної консолі, не може використовуватися на жодному іншому пристрої без серйозних зусиль від програмістів, щоб адаптувати ігрову програму до різного обладнання. Для комп'ютерних ігор цей ефект був ще гіршим. На сьогоднішній день операційні системи забезпечують *апаратну абстрактність,* тому програми не повинні мати справу з усіма різними типами апаратного забезпечення, які можуть бути встановлені на комп'ютері.Перш ніж це існувало, кожній грі потрібно було надавати власні драйвери для кожної відеокарти та звукової карти; в результаті, не так багато коду, написаного для певної гри, можуть бути використані повторно для іншої гри. В 1980s

Аркадні ігри були надзвичайно популярні, але майже ніхто з коду, написаного для них, не можна було повторно використовувати для нових ігор через постійні зміни та вдосконалення комп'ютерної техніки.

У міру того, як ігри стали більш складними, а операційні системи стали більш незалежними від апаратного забезпечення, для ігрових компаній стало сенсом почати повторне використання коду з попередніх ігор. Чому пишеш абсолютно нову програму візуалізації або програму перевірки зіткнень для кожної гри, якщо ви просто можете використовувати

один із раніше випущеної гри? Термін *ігровий движок* був створений у 1990-х роках, коли шутери від першої особи, такі як Doom і Quake, стали дуже популярним жанром. Ці ігри були настільки популярні, що їх виробник, id Software, вирішив ліцензувати частину ігрового коду іншим ігровим компаніям як окрему частину програмного забезпечення. Перепродаж основного ігрового коду як ігрового движка був вигідним заходом, оскільки інші компанії готові заплатити багато грошей за ліцензію на використання двигуна для власних ігор. Ці компанії більше не повинні писати свій власний ігровий код з нуля - вони можуть повторно використовувати програми, що містяться в ігровому движку, і зосереджуються більше на графічних моделях, персонажах, рівнях тощо. на

Сьогодні доступні багато різних движок гри .Деякі ігрові движки створені спеціально для такої платформи, як ігрова консоль або операційна система. Інші ігрові движки можуть використовуватися на різних платформах без необхідності змінювати програми, які використовують код двигуна гри. Це особливо корисно для ігрових компаній, які хочуть публікувати свої ігри різними платформи.

Сучасні ігрові движки забезпечують багато функціональних можливостей для розробників ігор, таких як 2D та 3D-рендеринг двигуна, спеціальні ефекти, такі як частинки та освітлення, звук, анімація, штучний інтелект, сценарії та багато іншого. Ігрові движки часто використовуються, тому що, розробляючи всі ці різноманітні інструменти займає багато роботи і тому ігрові компанії вважають за краще вкласти такий час і зусилля у створення красивих середовищ і складних рівнів.

Через це суворе відокремлення основних ігрових функцій та самої гри (рівнів, персонажів тощо), багато ігрових компаній наймають більше

художників, ніж програмісти. Проте програмісти все ще необхідні для покращення коду ігрового коду, а також для написання програм, які стосуються речей, які не входять до ігрового движка або які специфічні для гра

Крім того, ігрові компанії часто розробляють програмне забезпечення для підтримки розробки ігор, наприклад, програми редагування рівня, розширення програмного забезпечення 3D-моделювання для експорту моделей та анімацій у правильному форматі, інструменти для створення прототипів тощо.

Для JavaScript ще немає двигуна, який використовує кожен. Більшість людей програми відносно прості ігри в JavaScript, щоб переконатися, що ігри працюють на різних пристроях, особливо на пристроях з обмеженими можливостями. Тому замість того, щоб використовувати движок, програмісти називають гру безпосередньо за допомогою елементів HTML5, таких як полотно .Проте, це швидко змінюється. Якщо ви вводите *движок двигуна javascript* в Google, ви знайдете безліч двигунів, які можна використовувати як основу для розробки власних ігор.Мета цієї книги - навчити вас, як програмувати ігри; але ви не будете використовувати двигун, тому що я хочу навчити вас ядру мови та її можливостям. Це не керівництво для

ігровий движок. Фактично, прочитавши цю книгу, ви зможете створити свій власний ігровий движок. Я не кажу, що ви повинні це зробити, але ви зможете краще запрограмувати гру з нуля, і швидше зрозуміти, як бібліотека ігрового движка працює

Розробка ігор

Два підходи часто використовуються при розробці ігор. Рисунок 1-1 ілюструє ці підходи: зовнішня частина охоплює внутрішню. Коли люди вперше вивчають програму, вони зазвичай починають писати код негайно, що призводить до тісного циклу написання, а потім тестування, а потім внесення змін. Проте професійні програмісти, навпаки, проводять значний початковий час, роблячи проектну роботу, перш ніж писати першу рядок коду.

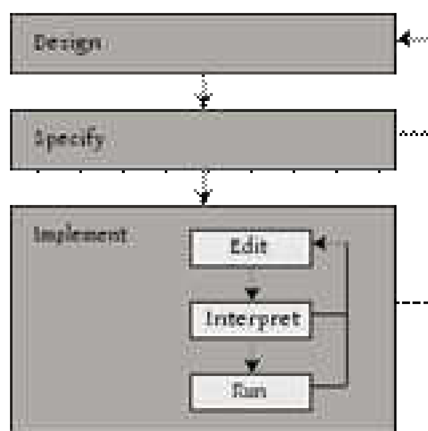


Рисунок 1.1 - Screen shot: Програмування в невеликих масштабах і у великому масштабі

Мала шкала: Edit-Interpret-Run

Якщо ви хочете створити гру в JavaScript, вам потрібно написати програму, яка містить багато рядків інструкцій. За допомогою текстового редактора ви можете редагувати скрипти, на яких ви працюєте. Після завершення запису цих інструкцій ви запустите браузер (бажано останню версію часто використовуваної програми браузера) і спробуйте запустити програму. Коли все добре, браузер інтерпретує сценарій і виконує його.

Проте в більшості випадків речі не так просто.З одного боку, вихідний код, який ви надаєте браузеру / перекладачеві, повинен містити дійсний код JavaScript, оскільки ви не можете очікувати, що браузер виконуватиме скрипт, що містить випадкове привласнення. Браузер перевіряє, чи відповідає вихідний код мовним специфікаціям мови JavaScript. Якщо ні, це спричиняє помилку, і сценарій зупиняється.Звичайно, програмісти докласти зусиль , щоб писати правильні програми JavaScript, але це легко зробити помилку, а також правила написання коректних програм є дуже суворими.Таким чином, ви, напевно, стикаєтесь з помилками під час інтерпретації фаза

Після кількох ітерацій, під час яких ви вирішуєте незначні помилки, браузер тлумачить весь сценарій без проблем. Наступним кроком браузер *виконує* або *запускає* сценарій.У багатьох випадках ви дізнаєтеся, що скрипт точно не робить те, що ви хочете зробити. Звичайно, ви доклали зусиль, щоб правильно висловити те, що ви хотіли зробити сценарій, але концептуальні помилки легко зробити.

Тому ви повернетесь до редактора, і ви зміните сценарій.Потім ви знову відкриваєте браузер і намагаєтесь інтерпретувати / запустити скрипт і сподіваємося, що ви не зробили нових помилок введення. Ви можете виявити, що попередня проблема вирішена, лише усвідомлюючи, що, хоча сценарій робить щось інше, він все одно не робить саме те, що ви хочете.І він повернувся до редакції знову.Ласкаво просимо до життя як програміст!

Велика шкала: Design-Specify-Implement

Як тільки ваша гра стає складнішою, вам не доведеться просто починати друкувати, поки не закінчите. Перш ніж почати *реалізацію* (написання та тестування гри), існують ще дві фази.

По-перше, вам потрібно *спроектувати* гру.Який тип гри ви будуєте? Хто така цільова аудиторія вашої гри? Це 2D-гра або 3D-гра? Який геймплей ви хотіли б моделювати? Які персонажі знаходяться в грі та які їх можливості? Особливо

коли ви розробляєте гру разом з іншими людьми, вам потрібно написати якийсь проектний документ, який містить всю цю інформацію, щоб кожен погодився на те, в якій грі вони розвиваються! Навіть коли ви розробляєте гру самостійно, це хороша ідея писати вниз

дизайн гри. Етап *проектування* насправді одна з найскладніших завдань розробки ігор.

Після того, як це ясно , що гра повинна робити, то наступний крок полягає

в створенні глобальної структури для програми.Це називається фаза *специфікації* .Ви пам'ятаєте, що парадигма об'єктно-орієнтованого програмування організовує інструкції в методах і методах у класах? У специфікації фаза ви зробити ан огляд від в заняття необхідний за в гра і в методи в ті заняття В це етап ви тільки треба до опишіть що а метод воля робити ні як його зроблено. Проте пам'ятайте, що не можна очікувати від методів неможливе: вони повинні бути реалізовані пізніше.

Коли специфікація гри закінчена, ви можете почати фазу *реалізації,* яка зазвичай означає , що проходить через редагування-інтерпретує запустити цикл кілька разів. Після цього ви можете дозволити іншим грати в вашу гру.

Лекція 2
Основні елементи ігор мовою JavaScript

Анотація. Лекція знайомить студентів з основними елементами ігор мовою JavaScript, зі структурою ігрової програми, макетом.

Мета лекції:

- Навчити студентів структурувати ігрову програму.
- Навчити студентів дотримуватисьправил макетування програм, розміщення коментарів.

2.1 Ігровий світ

Те, що робить ігри такою гарною формою розваг, полягає в тому, що ви можете вивчити уявний світ і робити те, що ви ніколи не зробите в реальному житті. Ви можете кататися на спині дракона, знищити цілі сонячні системи або створити складну цивілізацію персонажів, які говорять на уявній мові. Це уявне царство, в якому ви граєте в гру, називається *ігровим світом* .Світові ігри можуть варіюватися від дуже простих доменів, таких як світ Тетрісу, до складних віртуальних світів у таких іграх, як Grand Theft Auto та World of Warcraft.

Коли гра запущена на комп'ютері або смартфоні, пристрій підтримує внутрішнє уявлення ігрового світу.Це уявлення нічого подібного до того , що ви бачите на екрані , коли ви граєте в гру не дивитися.Вона складається в основному з чисел , що описують розташування об'єктів, скільки хитпоинтов противник може прийняти від гравця, скільки елементів гравець має в інвентарі, і так далі.До щастя, ця програма також уміє створювати візуально приємне уявлення цього світу , що він відображає на екрані.Інакше, грати в комп'ютерні ігри, ймовірно, буде неймовірно нудним, коли гравці повинні просіяти сторінки номерів, щоб з'ясувати, врятували вони принцесу або померли жахливою смертю.

Гравці ніколи не побачать внутрішнє уявлення ігрового світу, але

розробники ігор створюють його. Якщо ви хочете розробити гру, ви також повинні розробити , як представити свій ігровий світ всередині. І частина задоволення від програмування власних ігор є те , що у вас є повний контроль над цим.

Інша важлива річ, яку потрібно усвідомити, полягає в тому, що, як і в реальному світі, ігровий світ постійно змінюється. Монстри пересуваються в різні місця, погода змінюється, автомобіль вичерпується газом, вороги забиваються і так далі. Крім того, гравець дійсно впливає на зміну ігрового світу! Так що просто зберігати уявлення про ігровий світ в пам'яті комп'ютера недостатньо.

Гра також повинна постійно реєструвати те, що робить гравець, і, як наслідок, *оновлювати* це уявлення.Крім того, гра повинна *показати* світ ігор гравця, відображаючи його на моніторі комп'ютера, на екрані телевізора або на екрані смартфона.Процес, що стосується всього цього, називається *грою петля.*

2.2 Цикл гри

Цикл гри стосується динамічних аспектів гри. Багато чого трапляється під час гри. Гравці натискають кнопки на геймпад або торкаються екрана свого пристрою, і а

постійно мінливий ігровий світ, що складається з рівнів, монстрів та інших персонажів, повинен постійно змінюватися. Є також спеціальні ефекти, такі як вибухи, звуки та багато іншого. Всі ці різні завдання, які потрібно обробляти циклом гри, можуть бути організовані у два категорії:

- Завдання, пов'язані з оновленням та підтримкою ігрового світу
- Завдання, пов'язані з відображенням ігрового світу до гравця

Цикл гри постійно виконує ці завдання один за одним.Як приклад, давайте розглянемо, як ви могли б обробляти користувальницьку навігацію в простій грі, як Pac-Man.Ігровий світ в основному складається з лабіринту, де рухаються кілька неприємних примар. Pac-Man знаходиться десь у цьому лабіринті і рухається в певному напрямку. У першому завданні (оновлення і підтримка ігрового світу), ви перевіряєте, чи гравець натискає клавішу зі стрілкою .Якщо так, то вам потрібно оновити позицію Pac-Man відповідно до напряму, який хоче грати Pac-Man. Також, з-за цього руху, Pac-Man може з'їсти білу точку, що збільшує рахунок. що вам потрібні.

щоб перевірити, чи це останній крапка на рівні, тому що це означає, що гравець закінчив рівень. Нарешті, якщо це більша біла точка, примари повинні бути вимкнені.Потім потрібно оновити решту ігрового світу. Потрібно оновити становище привидів, ви повинні вирішити, чи повинні фрукти відображатися десь на бонусні очки, потрібно перевірити, чи Пак-Людина зіткнеться з одним з привидів (якщо привид не є бездіяльним), і так на Ви можете бачити, що навіть в простій грі, як Pac-Man, у цьому першому завданні потрібно багато роботи.Відтепер я буду дзвонити

це збірка різних завдань, пов'язаних з оновленням та підтримкою ігрового

світу Action Update .

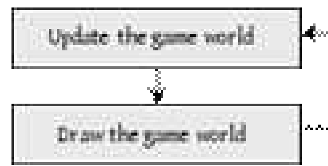


Рисунок 2.1 - Гра, яка постійно оновлює, а потім малює ігровий світ

Друга колекція завдань пов'язана з відображенням ігрового світу для гравця.У випадку від гри Pac-Man, це означає, що намалювати лабіринт, привидів, Pac-Man і інформацію про гру, важливу для гравця, щоб знати, наприклад, скільки очок вони забили, скільки їх залишили , і так далі.Ця інформація може відображатися в різних областях гри, наприклад, у верхній або нижній частині екрана. Ця частина дисплея також називається дисплеєм "*heads-up"* (HUD).Сучасні 3D-ігри мають набагато більш складні набори завдань малювання. Ці ігри повинні мати справу з освітленням і тіні, відбитками, очищенням, візуальними ефектами, такими як вибухи та багато іншого. Я назву частину ігрової петлі, яка стосується всіх завдань, пов'язаних з відображенням ігрового світу для гравця Draw дія.

2.3 Створення додатка для гри в JavaScript

В програмі мовою JavaScript ви бачили, що інструкції згруповані у функції, як показано нижче:

```
function changeBackgroundColor () {
document.body.style.background = "blue";
}
```

Ця ідея групування узгоджується з думкою, що JavaScript є процедурною мовою: інструкції згруповані в процедурах / функціях. Першим кроком є створення простої ігрової петлі в JavaScript. Подивіться на наступний приклад:

```
var canvas = undefined;
var canvasContext = undefined;

function start () {
canvas = document.getElementById ("myCanvas"); canvasContext =
canvas.getContext ("2d"); mainLoop ();
}
document.addEventListener ('DOMContentLoaded',start); function update () {
}

function draw () {
```

```
}

function mainLoop () (canvasContext.fillStyle = "blue";
canvasContext.fillRect (0, 0, canvas.width, canvas.height); update ();
update();
draw();
window.setTimeout (mainLoop, 1000/60);
}
```

Як видно, існує декілька різних функцій у цьому сценарії. Функція start викликається, коли тіло HTML-документа завантажено, через цю інструкцію:

```
document.addEventListener ('DOMContentLoaded', start);
```

У функції start ви отримуєте контекст полотна та саме полотно; ви зберігаєте їх у *змінних,* щоб ви могли їх використовувати в інших частинах програми (докладніше про це пізніше).Потім ви *виконуєте* ще одну функцію під назвою mainLoop .Ця функція, у свою чергу, містить інші інструкції. Дві update , після чого виконується функція draw .Кожна з цих функцій може знову містити інші інструкції. Остання інструкція, яка викликається є наступна:

```
window.setTimeout (mainLoop, 1000/60);
```

Що це робить - це просто викликає функцію mainLoop знову після очікування певного періоду часу (1000/60 = 16,6 мілісекунди в цьому випадку).Коли функція MainLoop викликається знову, колір фону встановлюється і update і draw функції викликаються. На даний момент update і draw порожні, але ви можете почати заповнювати їх з інструкціями , щоб оновити і зробити ігровий світ.Зауважте, що використання setTimeout, щоб почекати між ітераціями циклу, не завжди є найкращим рішенням. Інколи такий підхід може негативно вплинути на події поза вашим контролем, наприклад, на повільних комп'ютерах, інших вкладках, відкритих у вашому браузері, одночасно запускати програми, які потребують потужності обробки тощо. Коли вам доводиться мати справу з чутливими операціями часу (наприклад, плеєр, який потребує виживання протягом п'яти хвилин), ви можете не захотіти покладатися на setTimeout, а скоріше на якусь систему, яка розраховує події у певні моменти часу та перевіряє в update функція, чи є ці події сталося

Під час запуску програми з прикладом функції update та draw безперервно виконуються: оновлювати, малювати, оновлювати, малювати, оновлювати, малювати, оновлювати, малювати, оновлювати, малювати тощо. Крім того, це відбувається на дуже високій швидкості. Цей конкретний приклад створює просту гру, яка працює приблизно в 60 кадрів на секунду. Цей тип циклу називається *фіксованим кроком* циклу, і це дуже популярний вид циклу для казуальних ігор.Ви також можете спроектувати програму по-різному, щоб гра намагалася виконати цикл як можна більше разів, а не 60 разів на секунду.

Ця книга показує вам багато різних способів заповнення update та draw функцій з завданнями, які вам потрібно виконати у вашій грі.У ході цього процесу я також представляю багато методів програмування, корисних для ігор (та інших додатків). У наступному розділі докладніше розглянута основна ігрова програма. Потім ви заповнюєте цей основний каркас гри додатковими інструкціями.

2.4 Структура програми

У цьому розділі більш детально розповідається про структуру програми. У перші дні багато комп'ютерних програм писали тільки текст на екран і не використовували графіку. Така текстова програма називається *консольним* додатком.Окрім друку тексту на екрані, ці додатки також можуть читати текст, який користувач ввів на клавіатурі. Таким чином, будь-яке спілкування з користувачем здійснювалося у формі послідовностей запитань / відповідей (Ви хочете форматувати жорсткий диск (Y / N)? Ви впевнені (Y / N)? і так далі). Перед тим, як Windows- популярні ОС стали популярними, цей текстовий інтерфейс був дуже поширеним для текстових програм, електронних таблиць, математичних програм і навіть ігор.Ці ігри називались *текстовими пригодами* , і вони описували ігрове поле в текстовій формі.Гравець може потім ввести команди, щоб взаємодіяти з ігровим світом, наприклад, перейти на захід , підібрати матчі або Xyzzy .

Приклади таких ранніх ігор - Zork і Adventure. Хоча тепер вони здаються застарілими, вони все ще весело грають!

До цих пір можна писати консольні програми, також на такій мові, як JavaScript. Хоча цікаво подивитися, як писати такі програми, я предпочитаю зосередити увагу на програмуванні сучасних ігор з графікою.

Типи програм

Консольна програма є лише одним прикладом типу програми. Ще одним дуже поширеним типом є програма *Windows* .Таке додаток показує екран, що містить вікна, кнопки та інші частини *графічного інтерфейсу користувача* (GUI).Цей тип програми часто *керується подіями* : він реагує на такі події, як натискання кнопки або виділення елемента меню.

Інший тип програми - це *програма* , запущена на мобільному телефоні або планшетному ПК. Простір екрану, як правило, обмежено в цих типах програм, але доступні нові можливості взаємодії, такі як GPS, щоб дізнатись про місцезнаходження пристрою, датчики, які визначають орієнтацію пристрою, і сенсорний екран.

Під час розробки додатків досить складно написати програму, яка працює на всіх різних платформах. Створення додатка Windows сильно відрізняється від створення додатка. І повторне використання коду між різними типами програм важко. З цієї причини, *веб-додатки*

стають все більш популярними. У цьому випадку програма зберігається на

сервері, і користувач запускає програму в веб-браузері. Є багато прикладів таких застосувань: думайте про веб-програми електронної пошти або сайти соціальної мережі. І в цій книзі ви дізнаєтеся, як розвивати *веб-ігри* .

Функції

Пам'ятайте, що в імперативній програмі *інструкції* виконують справжню роботу програми: вони виконуються один за іншим.Це змінює пам'ять та / або екран, щоб користувач помітив, що програма робить щось. У програмі BasicGame не всі рядки в програмі є інструкціями. Одним з прикладів інструкції є лінія context.fillRect (0, 0, canvas.width, canvas.height); який наказує полотну малювати прямокутник на екрані кольором, вказаним у попередній інструкції.Через те, що цей прямокутник - це розмір полотна, весь колір полотна змінюється.

Оскільки JavaScript є процедурною мовою, інструкції можна згрупувати у *функції* .У JavaScript не обов'язково, щоб інструкція була частиною функції. Наприклад, наступна інструкція в програмі BasicGame не належить до функції:

var canvas = undefined;

Однак функції дуже корисні. Вони запобігають дублюванню коду, оскільки інструкції знаходяться лише в одному місці, і вони дозволяють програмісту легко виконувати ці інструкції, викликаючи одне ім'я. Інструкції групування у функції виконуються фігурними дужками ({ і }).Такий блок інструкцій, згрупованих разом, називається *тілом* функції.Над тілом ви пишете *заголовок* функції.Приклад заголовка функції виглядає наступним чином:

function mainLoop ()

Заголовок містить, серед іншого, *назву* функції (у цьому випадку mainLoop). Як програміст, ви можете вибрати будь-яке ім'я для функції.Ви бачили, що ігрова петля складається з двох частин: оновлення та малюнок .У термінах програмування ці частини моделюються як функції, як ви можете побачити в прикладі програми. У цих функціях ви потім поміщаєте інструкції, які ви хочете виконати, щоб оновити або намалювати світ ігор. Назву функції передує функція слова , а після назви - пара круглих дужок.Вони служать для надання інформації інструкціям, які виконуються всередині функції. Наприклад, подивіться на наступне заголовок:

function playAudio (audioFileId)

У цьому заголовку назва функції - playAudio ; і між дужками ви бачите слово audioFileId .Очевидно, функція playAudio вимагає ідентифікатора аудіофайлів, щоб він знав, який аудіо файл повинен відтворюватися.

Синтаксис Діаграми

Програмування на такій мові, як JavaScript, може бути складним, якщо ви не знаєте правил мови .Ця книга використовує так звані *синтаксичні діаграми,* щоб пояснити структуру мови.*Синтаксис* мови програмування відноситься до формальних правил , які визначають , що є допустимим програма (іншими словами: програма , що компілятор або інтерпретатор може прочитати).Навпаки, *семантика* програми посилатися до справжнього *значення* цього.Щоб проілюструвати різницю між синтаксисом та семантикою, подивіться на фразу "вся ваша база належить нам". Синтаксично ця фраза не є дійсною (інтерпретатор англійської мови точно скаржиться на це). Проте *сенс* цієї фрази цілком очевидний: ви, очевидно, втратили всі свої бази на чужої раси, яка говорить погано англійською.

Перекладач може перевірити синтаксис програми: будь-яка програма, яка порушує правила, відхилена.На жаль, перекладач не може перевірити, чи семантика програми відповідає тому, що мав на увазі програміст. Отже, якщо програма є синтаксично коректною, це не гарантує, що це семантично правильне.Але якщо це навіть не синтаксично правильно, воно не може працювати зовсім. Синтаксичні діаграми допомагають вам візуалізувати правила мови програмування, такі як JavaScript. Наприклад, малюнок 2.2 - спрощена діаграма синтаксису, яка показує, як визначити функцію в JavaScript

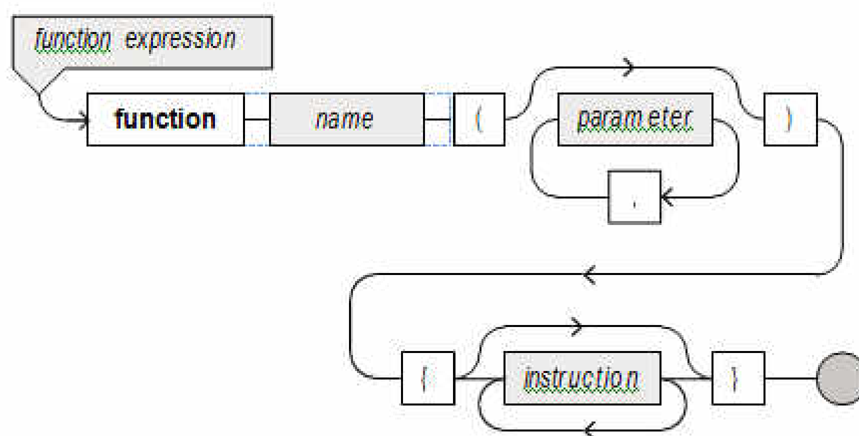


Рисунок 2.2 - Синтаксична діаграма для вираження функції

Ви можете використовувати синтаксичні діаграми для побудови коду JavaScript, починаючи з лівого верхнього краю діаграми, в даному випадку за допомогою *функції "* слово" та за допомогою стрілок.Коли ви досягаєте сірої точки, ваш фрагмент коду завершено. Тут ви можете чітко бачити, що визначення функції починається з ключового слова function ; тоді ви напишете назву функції.Після цього ви записуєте дужки. Між цими дужками ви можете (необов'язково) записати будь-яку кількість *параметрів* , розділених комами.Далі ви напишете декілька інструкцій, все між дужками. Після цього ви закінчите, оскільки ви досягли сірої точка. У цій книзі я використовую

синтаксичні діаграми, які показують, як структурувати код відповідно до синтаксичних правил JavaScript мова.

Виклик функції

Коли команда canvasContext.fillRect (0, 0, canvas.width, canvas.height); виконується, ви *викликаєте* функцію fillRect .Іншими словами, ви хочете, щоб програма виконувала команди, згруповані у функції fillRect .Ця група інструкцій робить саме те, що вам потрібно для цього прикладу: саме заповнення прямокутника кольором.Тим не менш, вам потрібно надати додаткову інформацію для цієї функції, оскільки вона повинна знати розмір прямокутника, який повинен бути заповнений.Параметри надають цю додаткову інформацію. Функція може мати більше одного параметра, як ви бачили на діаграмі синтаксису. Коли викликається функція, ви завжди записуєте круглі дужки за нею, а в дужках - це параметри (якщо вимагається).

Вам потрібно знати, які інструкції згруповані разом у функції fillRect , щоб використовувати їх?Ні, ти не робиш! Це одна з приємних речей щодо групування інструкцій у функціях. Ви (або інші програмісти) можете використовувати цю функцію, не знаючи, як вона працює.За жваво групування інструкції по функціям, можна написати повторно використовувані фрагменти програми , які можуть бути використані в самих різних контекстах.Функція fillRect є гарним прикладом цього.Він може бути використаний для різних додатків, і вам не потрібно знати, як ця функція працює, щоб використовувати її. Єдине, що вам потрібно знати, - це те, що він приймає розміри прямокутника як параметри

Оновлення та малювання

Ігрова петля у прикладі BasicGame містить функції update та draw.Оскільки функція є в основному групою інструкцій, кожен раз, коли викликається функція update, виконуються інструкції в цих функціях.Те ж саме стосується draw.

Як приклад, уявіть, що вам потрібна проста гра, в якій повітряна куля втягується в позицію вказівника миші. Коли ви рухаєте мишею, куля рухається разом з ним. З точки зору функцій update та draw, ви можете зробити це наступним чином.У функції update ви повинні виконати інструкцію, яка витягує поточну позицію вказівника миші та зберігає її в пам'яті.

У функції draw ви повинні виконати інструкцію, яка відображає кульовий образ у збереженій позиції.Звичайно, ви ще не знаєте, чи є ці інструкції (спойлер: вони роблять!), І ви ще не знаєте, як виглядають інструкції. Крім того, ви можете задатися питанням, чому це буде працювати. Ви не пересуваєте повітряну кулю, ви просто малюєте його на позиції, що зберігається в функції update .Нагадаємо, що update і draw функції виконуються з дуже високою швидкістю (60 разів на секунду).Оскільки

з цієї високої швидкості, намалюючи кулю на різних позиціях, робить її схожим на кулі (хоча насправді це не відбувається). Ось як складаються всі

ігрові світи і як гравця заманюється

мислення, є рух у світі. Насправді, ви просто малюєте зображення швидко на різних позиціях. Залиштеся налаштовано - ви повернетеся до цього прикладу і зробите його роботу пізніше!

Лекція 3
Реагування на дії користувача у грі мовою JavaScript.

Анотація. Лекція знайомить студентів з організацією руху спрайту, який повторює рух покажчика миші. Пояснюється створення обробника події руху миші. Пояснюється динаміка спрайту як реакція на подію руху миші.

Мета лекції:

- Ознайомити студентів з базовим прийомом отримання позиції покажчика миші.
- Сформувати у студентів навички створення динамічних спрайтів, які повторюють рух покажчика миші.

3.1 Об'єкти в іграх

До цих пір у всіх прикладних програмах був один великий об'єкт під назвою Game .Цей об'єкт складається з ряду змінних для зберігання полотна та його контексту, спрацьовування, позицій тощо. Це те , що об'єкт з прикладу Painter1 game виглядає наступним чином :

```
var Game = {
canvas : undefined,
canvasContext : undefined,
backgroundSprite : undefined,
cannonBarrelSprite : undefined,
mousePosition : { x : 0, y : 0 },
cannonOrigin : { x : 34, y : 34 },
cannonRotation : 0
};
```

Як ви бачите, він вже містить досить багато змінних, навіть для простої програми, яка малює лише фон та обертається гармату. Оскільки ігри, які ви розвиваєте, ускладнюються, цей список змінних буде рости, і в результаті код стане важче для інших розробників до зрозуміти (і для вас, коли ви не дивитесь на код на кілька місяців). Проблема в тому, що ви зберігаєте все в одному, великому об'єкті, званому Game .Концептуально це має сенс, тому що гра містить все, що стосується гри "Художник". Проте, код буде легше зрозуміти, якщо мало що відокремлювати.

Якщо ви дивитеся на вміст об'єкта Game , ви можете побачити, що певні змінні відносяться до певної міри.Наприклад, canvas та canvasContext змінні співвідносяться, тому що вони обидві стосуються полотна. Крім того, досить багато змінних зберігають інформацію про гармат, таких як його положення

або його обертання. Ви можете групувати пов'язані змінні в різні об'єкти, щоб переконатись, що ці змінні пов'язані ясніше в коді. Наприклад, подивіться на цей приклад:

```
var Canvas2D = {
canvas : undefined, canvasContext : undefined
};

var Game = {
backgroundSprite : undefined,
};

var cannon = {
cannonBarrelSprite : undefined, position : { x : 72, y : 405 },
origin : { x : 34, y : 34 }, rotation : 0
};

var Mouse = { position : { x : 0, y : 0 } };
```

Як ви бачите, у вас тепер є пара різних об'єктів, кожна з яких містить деякі змінні, які раніше були згруповані в об'єкті game.Тепер стало набагато легше побачити , які змінні відносяться до гармати і змінні , які належать до полотна.І приємно, що ти можеш це зробити в те ж саме для *методів*. Наприклад, ви можете додати методи очищення полотна та нанесення на неї зображення на об'єкт Canvas2D наступним чином:

```
Canvas2D.clear = function ()
{ Canvas2D.canvasContext.clearRect(0, 0, this.canvas.width,
this.canvas.height);
};

Canvas2D.drawImage = function (sprite, position, rotation, origin) {
// canvas drawing code
};
```

Використання різних об'єктів, на відміну від одного об'єкта, який містить все, що належить до гри, робить ваш код набагато простішим для читання. Звичайно, це справедливо лише в тому випадку, якщо ви розподіляєте змінні над об'єктами *логічно* .Навіть для простих ігор існує безліч способів упорядкування коду. Всі розробники мають свій власний стиль. Коли ви читаєте, ви дізнаєтеся, що ця книга також має певний стиль. Ви можете не погодитися з цим стилем, або іноді, можливо, ви подолали проблему інакше, ніж ми робимо в цій книзі.Нічого страшного. Існує майже ніколи не єдине правильне рішення для програмування проблеми.

Повертаючись до розподілу над об'єктами, ви можете побачити, що ми назвали більшість об'єктів, починаючи з символів верхнього регістру (наприклад, Canvas2D), але об'єкт гармати починається з символу нижнього регістру .Ми зробили це з причини, про яку ми обговорюємо більш детально пізніше.Поки що , скажемо, що об'єкти, що почати з ан великі літери характер є корисний за *будь-який* гра але в об'єкти чий імена почати з малими літерами використовуються лише для *певної* гри.У цьому випадку ви можете уявити, що об'єкт Canvas2D міг ,ути використаний в будь-який HTML5 гра.

3.2 Завантаження спрайтів

Тепер у вашій грі є різні об'єкти, де ви завантажуєте спрайт? Ви можете завантажити всі символи в початковому методі об'єкта Game , але іншим варіантом є додавання подібного методу, наприклад, об'єкта гармати, і завантажте там спрайт, що належить до гармати.Який підхід краще?

Існує щось сказати для завантаження спрайтів, що належать до об'єкта гармати, в методі ініціалізації цього об'єкта.Таким чином, ви можете чітко бачити з коду, які спрайі належать до якогось об'єкту.Однак це також означає, що якщо ви повторно використовуєте одне і те ж зображення для різних ігрових об'єктів, вам доведеться завантажувати цей спрайт кілька разів.І для ігор, що запускаються в браузері, це означає, що браузер повинен завантажити файл зображення з сервера, що може зайняти деякий час.Кращим варіантом є завантаження всіх спрацьовувань, необхідних для гри, коли гра починається. І щоб чітко відокремити спріди від решти програми, ви зберігаєте їх у об'єкті, який називається спрайтами .Цей об'єкт оголошується у верхній частині програми як порожній об'єкт:

```
var sprites = {};
```

Ви заповнюєте цю змінну справами в методі Game.start .Для кожного спрайду, який ви хочете завантажити, ви створюєте об'єкт зображення, а потім встановлюєте його джерело на місце спрацьовування. Оскільки ви вже використовуєте досить багато різних спрацьовувань, ви завантажуєте ці спрайти з іншої папки активів, яка містить всі спрайти, що належать грі Painter. Таким чином, вам не потрібно копіювати ці файли зображень для всіх різних прикладів у книзі, що використовує ці спрайти. Це інструкції, які завантажують різні спрайми, необхідні для прикладу Painter2:

```
var spriteFolder = "../../assets/Painter/sprites/";
sprites.background = new Image();
sprites.background.src = spriteFolder + "spr_background.jpg";
sprites.cannon_barrel = new Image();
sprites.cannon_barrel.src = spriteFolder + "spr_cannon_barrel.png";
```

```
sprites.cannon_red = new Image();
sprites.cannon_red.src = spriteFolder + "spr_cannon_red.png";
sprites.cannon_green = new Image();
sprites.cannon_green.src = spriteFolder + "spr_cannon_green.png";
sprites.cannon_blue = new Image();
sprites.cannon_blue.src = spriteFolder + "spr_cannon_blue.png";
```

Ви використовуєте оператор + тут, щоб об'єднати текст.Наприклад, значення виразу spriteFolder + "spr_background.jpg" є "../../assets/Painter/sprites/spr_background.jpg " .Шлях до папки "спрайт" виглядає трохи складніше. Також повинні бути надані деталі (такі як кількість учасників зустрічі, тип отриманих відгуків, заплановані тершахти тощо)./../ bit означає, що ви рухаєте два каталоги в ієрархії. Це необхідно, оскільки приклади каталогів Painter2 та Painter2a не знаходяться на тому ж рівні, що й каталог assets .Ви зберігаєте ці зображення у змінних, які є частиною об'єкта sprites .Пізніше ви отримуєте доступ до цього об'єкту, коли вам потрібно знайти спрайт.Наступним кроком буде керування гравцем натискання клавіш.

3.3 Обробка кнопки Подія

У попередній главі ви бачили, як використовувати обробник подій, щоб прочитати поточну позицію миші. У дуже подібному випадку ви можете реагувати на події, коли гравець тримає клавішу на клавіатурі. Знову ж таки, ви це робите, визначаючи обробник подій. Вам потрібно зберегти збережену клавішу, щоб ви могли пізніше отримати доступ до неї і зробити щось із цією інформацією. Найпростіший спосіб зберігати який ключ був натиснути - це використовувати *коди ключів* .Код ключа - це, в основному, число, що представляє певну клавішу. Наприклад, пробіл може бути номером 13, або ключ A може бути номером 65.

Отже, навіщо використовувати ці конкретні номери для цих ключів, а не для інших? Оскільки *коди таблиць персонажів* стандартизовані, і різні стандарти виникли протягом багатьох років.

У 70-х роках програмісти вважали, що $2^6 = 64$ символу було б достатньо, щоб відобразити всі можливі символи, які вам потрібні: 26 символів, 10 чисел та 28 знаків пунктуації (кома, крапка з комою та ін.).Хоча це означало, що не було ніякого розрізнення між символами нижнього та нижнього регістру, це не було проблемою на той час.

У 1980-х роках люди використовували $2^7 = 128$ різних символів: 26 великих літер, 26 малих символів, 10 чисел, 33 знаків пунктуації та 33 спеціальних символів (закінчення рядка, табуляція, звуковий сигнал тощо).Порядок цих символів був відомий як *ASCII* : Американський стандартний код для обміну інформацією.Це було добре для англійської мови, але це було недостатньо для інших мов, таких як французька, німецька, голландська, іспанська та ін.

Як наслідок, в 1990-х роках нові кодові таблиці були побудовані з $2^{8} = 256$ символів; були представлені також найпоширеніші листи для різних країн.Символи від 0-127 були такими ж, як у ASCII, але символи 128-255 використовувалися для спеціальних символів, що належать до певної мови. Залежно від мови (англійська, російська, індійська тощо) використовувалася інша кодова таблиця. Західноєвропейська кодова таблиця була латиницею1, наприклад. Для Східної Європи була використана інша кодова таблиця (у польській та чеській мовах багато спеціальних акцентів, для яких в таблиці Latin1 не було більше місця). Грецький, російський, іврит та індійський алфавіт Девангарі мали власні кодовні таблиці. Це був розумний спосіб спілкування з різними мовами, але речі ускладнювалися, якщо ви хочете одночасно зберігати текст на різних мовах. Крім того, мови, що містять більше 128 символів (наприклад, мандарин), не могли бути представлені за допомогою цього формат

На початку двадцять першого століття стандарт кодування знову поповнився таблицею, що містить $2^{16} = 65536$ різних символів.Ця таблиця може легко містити всі алфавіти у світі, включаючи багато різних знаків пунктуації та інших символів. Якщо ви коли-небудь стикаються з чужорідними видами, ця таблиця, напевно, може опинитися на символах мови іноземця .Кодова таблиця називається *Unicode* .Перші 256 символів Юнікоду - це ті самі символи, що й код Latin1 стіл

Повертаючись до коду ключа, який ви хочете зберегти для прикладу, давайте додамо просту змінну, яка містить останню натиснуту клавішу:

```
var Keyboard = { keyDown : -1 };
```

Коли ініціалізована змінна, вона містить змінну keyDown, яка містить значення -1.Це значення означає, що програвач *наразі не натискає жодної клавіші* .Коли програвач натискає клавішу, вам слід зберегти код клавіші в змінній Keyboard.keyDown .Ви це робите, написавши *обробник події,* який зберігає натиснуту на даний момент клавішу.Ось що виглядає обробник обробки подій:

```
function handleKeyDown(evt)  { Keyboard.keyDown = evt.keyCode;
}
```

Як ви бачите, функція отримує подію як параметр. Цей об'єкт події має перемінну назву

keyCode , що містить ключовий код клавіші, яку на даний момент програвач натискає.

Ви призначаєте цю функцію обробника подій у Game.start наступним чином:

```
document.onkeydown = handleKeyDown;
```

Тепер, коли програвач натискає клавішу, код клавіші зберігається так, що

ви можете використовувати його в своїй грі.Але що трапляється, коли гравець випускає ключ? Значення Keyboard.keyDown слід буде призначено -1 ще раз, щоб ви знали, що програвач наразі не натискає жодної клавіші.Це робиться за допомогою *ключа до* обробника подій.Ось заголовок і тіло цього обробник:

```
function handleKeyUp(evt)  { Keyboard.keyDown = -1;
}
```

Як бачите, це дуже просто. Єдине, що вам потрібно зробити - призначити значення -1 для keyDown змінна в об'єкті клавіатури .Нарешті, ви призначаєте цю функцію в Game.start :

```
document.onkeyup = handleKeyUp;
```

Тепер ви готові справитися з натисканнями клавіш у вашій грі. Зауважте, що цей спосіб вирішення натискань клавіш трохи обмежений. Наприклад, неможливо відслідковувати одночасні натискання клавіш, наприклад, плеєр, натискаючи клавіші A і B одночасно. Пізніше, у розділі 13, ви поширюєте об'єкт клавіатури, щоб взяти це до уваги.

3.4 Умовне виконання

Як простий приклад того, як ви можете використовувати об'єкт Keyboard, щоб щось зробити, давайте зробимо розширення програми Painter1, яка малює кольоровий кульку на верхівці гармати.Натискаючи клавішу R, G або B, гравець може змінити гармати колір на червоний, зелений або синій. На рисунку 6-1 показано знімок екрана програми.

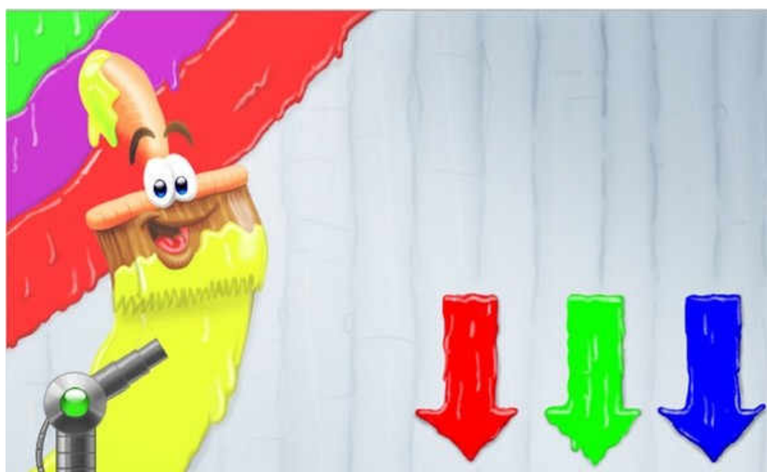


Рисунок 3.1 – Знімок екрана програми Painter2

Вам потрібно завантажити три додаткові спрайти, по одному для кожного кольорового кулі. Це робиться за допомогою трьох інструкцій:

```
sprites.cannon_red = Game.loadSprite(spriteFolder + "spr_cannon_red.png");
sprites.cannon_green = Game.loadSprite(spriteFolder + "spr_cannon_green.png");
sprites.cannon_blue = Game.loadSprite(spriteFolder + "spr_cannon_blue.png");
```

Ви додаєте метод initialize до об'єкта cannon , в якому ви призначаєте значення змінам, що належать цьому об'єкту.Цей метод викликається з Game.start .Таким чином, гармата ініціалізується, коли починається гра:

```
Game.start = function () { Canvas2D.initialize("myCanvas");

document.onkeydown = handleKeyDown;
document.onkeyup = handleKeyUp;
document.onmousemove = handleMouseMove;
...
cannon.initialize();  window.setTimeout(Game.mainLoop, 500);
};
```

У методі cannon.initialize ви призначаєте значення змінних, що належать до гармати.Це повний метод:

```
cannon.initialize = function() { cannon.position = { x : 72, y : 405 };
cannon.colorPosition = { x : 55, y : 388 };
cannon.origin = { x : 34, y : 34 };
cannon.currentColor = sprites.cannon_red; cannon.rotation = 0;
};
```

Як бачите, у вас є дві змінні позицій: одна для гармати та одна для кольорової кулі.Крім того, ви додаєте змінну, яка відноситься до поточного кольору сфери, яку слід намалювати. Спочатку ви призначаєте спрайт червоної сфери для цієї змінної.

Щоб чітко розподілити об'єкти, ви також можете додати метод draw в об'єкт cannon. У цьому методі ви малюєте гармату та кольорову сферу зверху:

```
cannon.draw = function () {
Canvas2D.drawImage(sprites.cannon_barrel, cannon.position, cannon.rotation, cannon.origin);
Canvas2D.drawImage(cannon.currentColor, cannon.colorPosition, 0,
{ x : 0, y : 0 });
};
```

Цей метод draw викликається з Game.draw наступним чином:

```
Game.draw = function () { Canvas2D.clear();
Canvas2D.drawImage(sprites.background, { x : 0, y : 0 }, 0,
```

```
{ x : 0, y : 0 });
cannon.draw();
};
```

Таким чином, ви можете побачити, які інструкції креслення належать до певного об'єкту.Тепер, коли підготовча робота виконана, ви можете розпочати обробку натискання клавіш гравця. До сих пір , всі інструкції ,які ви написали повинні були бути виконуватись весь час.Наприклад, программі завжди потрібно намалювати спрайти фону і гармати.Але тепер ви стикаєтеся з ситуацією, коли вам потрібно виконувати інструкції лише тоді, коли виконується певна умова. Наприклад, вам потрібно змінити колір кулі на зелений , *якщо гравець натискає кнопку G.Цей вид інструкцій називається умовною інструкцією* , і вона використовує нове ключове слово: if.

З , інструкцією if, ви можете забезпечити умову і виконати блок інструкцій, якщо ця умова виконана (в цілому, це іноді називають також *розгалуження).* Ось кілька прикладів умов:

- Гравець натиснув кнопку G
- Кількість секунд, що минули з початку гри, більше ніж 1000
- Спрайт кулі знаходиться саме в середині екрану
- Монстр з'їв твого персонажа

Ці умови можуть бути або *істинними,* або *хибними* .Умова є *виразом* , оскільки вона має значення (це або *true,* або *false*).Це значення також називається *логічним* значенням.З , інструкцією if, ви можете виконати блок інструкцій, якщо умова є істинною.Погляньте на цей приклад, інструкції if:

```
if (Game.mousePosition.x > 200) { Canvas2D.drawImage(sprites.background, {
x : 0, y : 0 }, 0,
{ x : 0, y : 0 });
}
```

Умова завжди розміщується в дужках. Наступним є блок інструкцій, вкладений в дужки. У цьому прикладі фон витягується, лише якщо позиція-х миші більше 200. В результаті, якщо ви переміщаєте мишу занадто далеко наліво на екрані, фон не малюється. Ви можете розмістити декілька вказівок між дужками, якщо хочете:

```
if (Game.mousePosition.x > 200) { Canvas2D.drawImage(sprites.background,
{ x : 0, y : 0 }, 0,
{ x : 0, y : 0 });
cannon.draw();
}
```

Якщо є лише одна інструкція, можна пропустити дужки, щоб трохи скоротити код:

```
if (Game.mousePosition.x > 200)
Canvas2D.drawImage(sprites.background, { x : 0, y : 0 }, 0,{ x : 0, y : 0 });
```

У цьому прикладі ви зміните колір гармати тільки тоді, коли гравець натискає клавішу R, G або B. Це означає, що ви повинні перевірити, чи наразі натискається одна з цих клавіш. Використовуючи об'єкт Keyboard , умова, яка перевіряє, чи натискається клавіша R, дається наступним чином:

```
Keyboard.keyDown === 82
```

Оператор === порівнює два значення і повертає true, якщо вони однакові або false в іншому випадку.

Зліва від цього оператора порівняння є значення змінної keyDown в об'єкті Keyboard .На правій стороні знаходиться код, що відповідає клавіші R. Тепер ви можете використовувати його в інструкції if наступним чином, в методі update, об`єкту cannon:

```
if (Keyboard.keyDown === 82) cannon.currentColor = sprites.cannon_red;
```

3.5 Оператори порівняння

У заголовку інструкції if є вираз, який повертає значення істини: *так* чи *ні* .Якщо результат виразу є « *так»* , виконується тіло інструкції if .У цих умовах вам дозволяється використовувати оператори порівняння. Доступні наступні оператори:

- < Менше ніж
- <= Менше або рівно
- > Більш ніж
- > = Більше або дорівнює
- === Дорівнює
- !== Не дорівнює

Ці оператори можуть використовуватися між будь-якими двома значеннями. З лівого та правої сторони цих операторів ви можете поставити постійні значення, змінні або повні вирази з додаванням, множенням або що завгодно. Ви перевіряєте рівність двох значень, використовуючи знак потрійної рівності (===).Це сильно відрізняється від одного знака рівно, що означає присвоювання. Різниця між цими двома операторами дуже важлива:

x = 5; означає: *призначити* значення 5 змінній x

x === 5 означає: *чи* x дорівнює 5?

Оператор == також порівнює значення, але якщо ці значення не мають того самого типу, цей оператор перетворює одне з таких значень, що відповідають типам. Таке перетворення звучить як наче може бути корисним, але це

призводить до деякої дивної поведінки системи. Ось кілька прикладів.

```
'' == '0'     // false
0 == ''       // true!
0 == '0'      // true!
```

Оператор === повернеться помилково в усіх трьох випадках, оскільки типи відрізняються. Як правило, краще уникати оператора ==. Оператор === є набагато більш передбачуваним, що призводить до зменшення кількості помилок та помилок, коли ви використовуєте його у вашій програмі.

3.6 Логічні оператори

У логічному плані умову також називають *предикатом* .Оператори, які використовуються в логіці для підключення предикатів (*and*, *or*, чи *not*), також можуть бути використані в JavaScript.Вони мають особливий характер позначення:

- && є логічним оператором *and*
- || є логічним оператором *or*
- ! є логічним оператором *not*

Ви можете використовувати ці оператори для перевірки складних логічних висловлювань, ви можете виконувати інструкції лише в особливих випадках. Наприклад, ви можете зробити вивід напису "Ти виграв!", тільки якщо у гравця більше 10 000 балів, ворог має життєву силу 0, а життєва сила гравця перевищує 0:

```
if (playerPoints > 10000 && enemyLifeForce === 0 && playerLifeForce > 0)
Canvas2D.drawimage(winningOverlay, { x : 0, y : 0 }, 0, { x : 0, y : 0 });
```

Булевий тип

Вирази, які використовують оператори порівняння або які з'єднують інші вирази з логічними операторами, також мають тип, як і вирази, які використовують арифметичні оператори. Зрештою, результат такого виразу є значенням: одне з двох значень істини - *так* чи *ні* .У логіці ці значення називаються *істинними* і *помилковими* .У JavaScript ці значення правди відображаються як ключові слова true і false.

Логічне вираження схоже на арифметичний вираз, за винятком того, що він має інший тип. Наприклад, ви можете зберігати результат логічного виразу в змінній, передавати його як параметр або знову використовувати цей результат в іншому виразі.

Тип логічних значень *булевий* , названий на честь англійського математика і філософа Джорджа Була (1815-1864).Ось приклад створення та присвоєння булевої змінної:

```
var test;
test = x> 3 && y <5;
```

Якщо x містить наприклад, значення 6 і y містить значення 3, Булевський вираз x> 3 && y <5 буде оцінюватися як true, і це значення буде збережено в змінній test . Ви також можете зберігати в булевій змінній значення truei false:

```
var isAlive = false;
```

Булеві змінні дуже зручні для зберігання стану різних об'єктів в грі.Наприклад, ви можете використовувати булеву змінну для того, щоб знати чи гравець все ще живий, чи гравець в даний час стрибає, чи завершений рівень, і так далі.Ви можете використовувати булеві змінні як вираз в
інструкції if:

```
if (isAlive)
// зробити щось
```

У цьому випадку, якщо вираз isAlive оцінює значення як true , виконується тіло команди if .Ви можете подумати, що цей код спричинить помилку компілятора, і вам потрібно зробити порівняння булевої змінної, як це:

```
if (isAlive === true)
// do something
```

Проте це додаткове порівняння не є необхідним. Вираз у інструкції if, *має оцінюватись як* true або false .Оскільки логічна змінна вже представляє одне з цих двох значень, вам не потрібно виконувати порівняння. Насправді, якщо попереднє порівняння було потрібне, вам також доведеться знову порівняти цей результат з логічним значенням:

```
if ((isAlive === true) === true)
// do something
```

Таким чином, не робіть речей складнішим, ніж вони є. Якщо результат вже є логічним значенням, вам не потрібно порівнювати його з чим-небудь.

Ви можете використовувати логічний тип для зберігання складних виразів , які дорівнюють або true , або false. Давайте подивимося на кілька додаткових прикладів:

```
var a = 12 > 5;
var b = a && 3 + 4 === 8;
var c = a || b;
```

```
if (!c)
a = false;
```

Перш ніж читати далі, спробуйте визначити значення змінних a, b, c після того, як ці інструкції були виконані.У першому рядку, ви заявляєте і змінюєте змінну a. Значення істини, яке зберігається в цьому Boolean обчислюється з виразу 12> 5, яке є істиною.Потім це значення присвоюється змінній a. У другому рядку, ви заявляєте і змінюєте нову змінну b, в якому ви зберігаєте результат більш складного виразу. Перша частина цього виразу є змінною a, яка містить істинне значення. Друга частина виразу є вираз порівняння 3 + 4 === 8. Це порівняння не відповідає дійсності (3 + 4 не дорівнює 8), так що це має значення FALSE, і , отже, логічне *і* також призводить до значення FALSE.Таким чином, змінна Ь містить значення FALSE після виконання цієї інструкції.

Третя команда зберігає результат логічної операції *або* на змінних a та b в змінній c .Оскільки a містить значення true , результат цієї операції також true , і цей результат присвоюється c.

Нарешті, є інструкція if , яка призначає значення false до змінної a , але тільки якщо !c є істиною .У цьому випадку, c є true, тому !c є FALSE, що означає , що тіло інструкції if не виконується.Тому, після того, як всі інструкції виконуються, обидва значення a та c містять значення true , a b містить значення FALSE.

Використання цих вправ показує, що дуже легко зробити логічні помилки. Цей процес подібний до того, що ви виконуєте, коли ви налагоджуєте код. Крок за кроком ви переходите через інструкції та визначаєте значення змінних на різних етапах. Одне змішування може спричинити те, що ви вважаєте true, може бути FALSE!

Націлювання зброї на вказівнику миші

У попередніх розділах ви бачили, як використовувати інструкцію if, щоб перевірити, чи гравець натиснув клавішу R.Тепер, припустимо, ви хочете оновити кут гармати тільки в тому випадку, якщо ліва кнопка миші натиснута. Для обробки натискання кнопки миші вам потрібно ще два оброблювачі подій: один для обробки події, коли користувач натискає кнопку миші, а інший - для обробки події, коли користувач випускає кнопку миші. Це робиться способом, аналогічним натисканню та випуску клавіші на клавіатурі. Кожного разу, коли кнопка миші натискається або випускається, змінна which в об'єкті події повідомляє вам, яка кнопка вона була (1 - ліва кнопка, 2 - середня кнопка, 3 - права кнопка).Ви можете додати логічну змінну до об'єкта Mouse, яка вказує на те, чи кнопка миші вниз.Давайте зробимо це для лівої кнопки миші:

```
var Mouse = {
position : { x : 0, y : 0 }, leftDown : false
};
```

Вам також потрібно додати дві функції обробника, які призначають значення для змінної leftDown .Ось дві функції:

```
function handleMouseDown (evt) {if (evt.which === 1)
Mouse.leftDown = true;
}

function handleMouseUp  (evt) {if (evt.which === 1)
Mouse.leftDown = false;
}
```

Як видно, ви використовуєте інструкцію if, щоб дізнатись, чи натиснута чи випущена ліву кнопка миші.Залежно від значення істини умови, ви виконуєте тіло інструкції. Звичайно, вам слід призначити ці обробники до відповідних змінних в документі, щоб вони були викликані при натисканні кнопки миші або її випуску:

```
document.onmousedown = handleMouseDown;
document.onmouseup = handleMouseUp;
```

Тепер, в методі update, обєкту cannon , ви оновлюєте кут гармати, якщо ліва кнопка миші натиснута:

```
if (Mouse.leftDown) {
var opposite = Mouse.position.y - this.position.y;
var adjacent = Mouse.position.x - this.position.x;
cannon.rotation = Math.atan2(opposite, adjacent);
}
```

Припустимо, ви хочете перезавантажити кут до нуля після того, як гравець відпустить ліву кнопку миші.Можна додати ще одну, інструкція if:

```
if (!Mouse.leftDown) cannon.rotation = 0;
```

Для більш складних умов це рішення буде складніше зрозуміти. Існує більш приємний спосіб вирішення цієї ситуації: за допомогою інструкції if,з *іншою альтернативою* .Альтернативна команда виконується, коли умова в інструкції if не відповідає дійсності; ви використовуєте ключове слово else для що:

```
if (Mouse.leftDown) {
var opposite = Mouse.position.y - this.position.y;
var adjacent = Mouse.position.x - this.position.x; cannon.rotation = Math.atan2(opposite, adjacent);
```

```
} else
cannon.rotation = 0;
```

Ця інструкція виконує точно так само, як і попередні дві, інструкції if, але вам потрібно лише написати умову один раз.Виконайте програму Painter2 і подивіться, що вона робить. Зверніть увагу, що кут гармати дорівнюватиме нулю, як тільки ви відпустите ліву кнопка миші.

Синтаксис інструкції if з альтернативою представлений синтаксичною діаграмою на рис 6-2.Тіло , інструкції if може складатися з декількох інструкцій між дужками , тому що інструкція також може бути *блоком* інструкцій, як це визначено в синтаксичної діаграмі на рисунку 3.2.

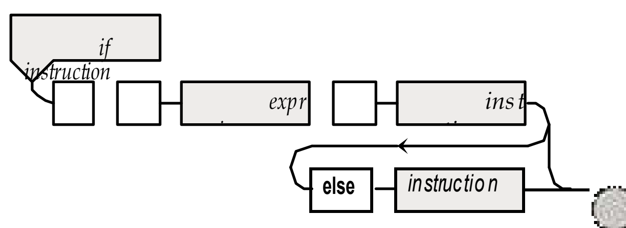


Рисунок 3.2 – Синтаксична схема інструкції if

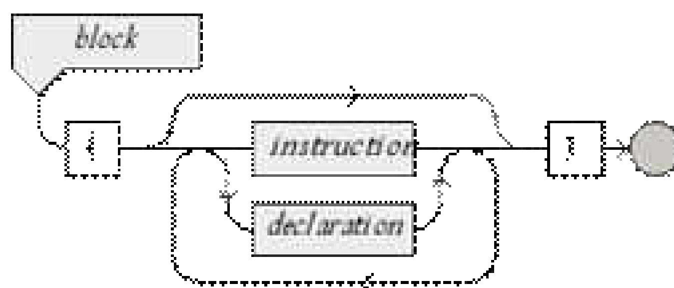


Рисунок 3.3 – Синтаксична схема блоку інструкцій (яка в свою чергу є інструкцією)

Декілька різних альтернатив

Якщо існує декілька категорій значень, ви можете дізнатись, з if з якими вказівками ви маєте справу. Другий тест розміщується після else першої інструкції if, так що другий тест виконується лише після першого тесту.Третій тест може бути поміщений після else частини, інструкції if, і так далі.

Наступний фрагмент визначає, в якому віці гравець потрапляє до гри, щоб ви могли намалювати різні спрайти гравця:

```
if (age < 4)
Canvas2D.drawImage(sprites.babyPlayer, playerPosition, 0,
{ x : 0, y : 0 });
else if (age < 12)
```

```
Canvas2D.drawImage(sprites.youngPlayer, playerPosition, 0,
{ x : 0, y : 0 });
else if (age < 65)
Canvas2D.drawImage(sprites.adultPlayer, playerPosition, 0,
{ x : 0, y : 0 });
else
Canvas2D.drawImage(sprites.oldPlayer, playerPosition, 0,
{ x : 0, y : 0 });
```

Після кожного else (крім останнього) є інша інструкція if.Для немовлят спрайт babyPlayer малюється, а решту інструкцій ігнорують (вони після цього else).З іншого боку, старі гравці проходять всі випробування (молодші за 4? молодше 12 років молодше 65 років?) перед тим, як зробити висновок, що вам доведеться намалювати спрайт oldPlayer.

Я використовував відступ у цій програмі, щоб вказати, який else належать до якого if .Коли існує багато різних категорій, текст програми стає все менш читабельним. Тому, як виняток з звичайного правилави можете використовувати більш простий макет:

```
if (age < 4)
Canvas2D.drawImage(sprites.babyPlayer, playerPosition, 0,
{ x : 0, y : 0 });
else if (age < 12)
Canvas2D.drawImage(sprites.youngPlayer, playerPosition, 0,
{ x : 0, y : 0 });
else if (age < 65)
Canvas2D.drawImage(sprites.adultPlayer, playerPosition, 0,
{ x : 0, y : 0 });
else
Canvas2D.drawImage(sprites.oldPlayer, playerPosition, 0, { x : 0, y : 0 });
```

Додатковою перевагою тут є те, що за допомогою цього макета, набагато простіше з'ясувати, які випадки обробляються інструкціями.Ви також можете побачити, що у прикладі коду використовується декілька альтернатив для обробки трьох різних кольорових випадків у методі update об'єкта cannon:

```
if (Keyboard.keyDown === Keys.R)
cannon.currentColor = sprites.cannon_red;
else if (Keyboard.keyDown === Keys.G)
cannon.currentColor = sprites.cannon_green;
else if (Keyboard.keyDown === Keys.B)
cannon.currentColor = sprites.cannon_blue;
```

Поряд з інструкцією if є інструкція, яка називається switch, який краще

підходить для вирішення багатьох різних альтернатив.Див. Розділ 21 додаткову інформацію про використання switch.

Переключення поведінки зброї

Як останній приклад використання інструкції if для обробки натискання кнопки миші, спробуємо натиснути кнопку *миші* замість затискання.Ви знаєте, як перевірити за допомогою інструкції if чи кнопка миші на даний момент затиснута, але як ви дізнаєтеся, чи *натиснув її* гравець (натиснувши кнопку, коли її раніше не натискали)?Подивіться на програму Painter2a. У цій програмі обертання гарматного ствола відбувається за напрямом руху миші після натискання лівої клавіші. Після повторного натискання гармата перестає слідувати за покажчиком миші.

Проблема з такою поведінкою *перемикання* полягає в тому, що ви знаєте лише поточний стан миші у методі update .Ця інформація недостатня для того, щоб з'ясувати, коли відбувається клік, оскільки клік частково визначається тим, що сталося під час попереднього перебування в методі update .Ви можете сказати, що гравець натиснув кнопку миші, якщо відбудуться ці дві речі:

- В даний час кнопка миші є натиснута
- Кнопка миші не була натиснута під час останнього виклику методу update

Ви додаєте додаткову логічну змінну leftPressed до об'єкта Mouse , яка вказує, чи відбулось натискання миші.Вам слід встановити цю змінну true, якщо ви отримуєте подію миші вниз (відповідаючи першій кулі), а змінна Mouse.leftDown ще не true (що відповідає другій кулі).Ось як виглядає обробник розширеного обробника подій handleMouseDown :

```
function handleMouseDown(evt) {
if (evt.which === 1) {
if (!Mouse.leftDown) Mouse.leftPressed = true;
Mouse.leftDown = true;
}
}
```

Тут показана *вкладена* інструкція if, в тілі інструкції містиняться одна або більше інструкцій if.Тепер ви можете написати код , необхідний для перемикання поведінки гармати, написавши , інструкцію if , яка перевіряє , чи була ліва кнопка миші натиснута:

```
if (Mouse.leftPressed)
cannon.calculateAngle = !cannon.calculateAngle;
```

У тілі інструкції if ви змінюєте calculateAngle .Це булева частина змінної об'єкта cannon.Для того, щоб отримати перемикач, ви використовуєте логічний оператор *not*.Результат операції *not* по змінної calculateAngle знову зберігається

в змінної calculateAngle.Отже, якщо ця змінна містить значення true , то в тій же змінній зберігається значення false і навпаки.Результат полягає в тому, що значення calculateAngle змінюється під час кожного виконання цієї інструкції.

Тепер ви можете використовувати цю змінну в іншій, інструкції if, яка визначає, чи слід оновлювати кут:

```
if (cannon.calculateAngle) {
var opposite = Mouse.position.y - this.position.y; var adjacent = Mouse.position.x - this.position.x; cannon.rotation = Math.atan2(opposite, adjacent);
} else
cannon.rotation = 0;
```

На даний момент змінна Mouse.leftPressed ніколи не скидається. Отже, після кожного виконання циклу гри ви скидаєте

Mouse.leftPressed до false .Ви додаєте метод reset до об'єкта Mouse , який це робить, таким чином:

```
Mouse.reset = function() { Mouse.leftPressed = false;
};
```

І, нарешті, цей метод викликається з методу mainLoop в об'єкті Game :

```
Game.mainLoop = function() { Game.update(); Game.draw(); Mouse.reset();
window.setTimeout(Game.mainLoop, 1000 / 60);
};
```

Лекція 4
Створення обробників подій натискання клавіш у ігрових веб-додатках мовою програмування JavaScript.

Анотація. Лекція знайомить з прийомами реагування на натискання клавіш. Розглядаються додаткові можливості структурування коду за допомогою об'єктів та методів.

Мета лекції:

- Ознайомити студентів з прийомами реагування на натискання клавіш.
- Ознайомити студентів зі способом підтримки події натискання клавіші.
- Довести до студентів додаткові можливості структурування коду за допомогою об'єктів та методів.

4.1 Використання окремих файлів JavaScript

Почнемо організовувати код гри Painter. Це необхідно, оскільки код гри містить багато рядків коду. У попередніх прикладах ви почали групувати змінні в різних об'єктах (наприклад, Canvas2D або cannon). Зараз продовжуєте структурувати свій код, використовуючи більше об'єктів і розділяючи код на окремі файли.

Можливо, ви помітили, що ваш JavaScript-файл стає досить великим. Мати один великий файл, який містить весь ваш код, є неправильним, оскільки це ускладнює пошук певних частин програми. Більш доцільним є розбиття коду на декілька вихідних файлів. Хороший спосіб зробити це - розділити різні об'єкти JavaScript на окремі файли JavaScript, коли кожен файл JavaScript міститиме код одного з об'єктів. Програма Painter3 містить об'єкти, але кожен об'єкт описаний у певному файлі JavaScript. Ви навіть можете помістити ці файли в окремі каталоги, щоб вказати, які об'єкти повинні бути разом. Наприклад, ви можете помістити файли JavaScript Keyboard і Mouse в каталозі з іменем input. Таким чином, стане ясно, що обидва ці файли містять код, що відноситься до роботи з вводом даних. Завантаження окремих файлів JavaScript до браузера є дещо складним. У попередніх прикладах ви завантажували файл наступним чином:

```
<script src = "FlyingSpriteWithSound.js"> </ script>
```

Ви можете подумати, що ви зможете завантажувати файли JavaScript, які використовуються в програмі Painter3, просто додаючи до HTML-файлу наступні елементи script:

```
<script src = "input / Keyboard.js"> </ script>
<script src = "input / Mouse.js"> </ script>
<script src = "Canvas2D.js"> </ script>
<script src = "system / Keys.js"> </ script>
<script src = "Painter.js"> </ script>
<script src = "Cannon.js"> </ script>
```

На жаль, ви зіткнетеся з проблемами, якщо ви намагаєтеся прийняти цей підхід, додавши більше і більше елементів script. Оскільки файли JavaScript завантажуються з сервера, то неможливо точно дізнатися, який файл JavaScript буде завантажений першим. Припустимо, що перший файл, завантажений браузером, - це Painter.js .Браузер не може інтерпретувати код в цьому файлі, так як код посилається на код в інших файлах (наприклад, об'єкт Canvas2D). І це стосується й інших файлів. Таким чином, для того, щоб це працювало, вам необхідно переконатися , що файли завантажуються в певному порядку, який підкорюється існуючим залежностям між файлами.

Іншими словами: якщо файл A потребує файл B, вам потрібно завантажити

файл B перед файлом A. У JavaScript можна змінювати HTML-сторінки; тому, теоретично, ви можете додати додатковий елемент сценарію до HTML-сторінки, яка потім почне завантажувати інший файл JavaScript. Використовуючи розумні способи обробки подій, ви могли б уявити собі написання коду JavaScript, який завантажує інші файли JavaScript у попередньо визначеному порядку.

Замість того, щоб писати весь цей код самостійно, ви також можете використовувати код, вже написаний іншими. Це ще одна перевага структурування коду: воно робить код більш корисним для інших програм.

4.2 Завантаження ігрових складових неправильним шляхом

Раніше ми говорили про завантаження файлів браузера в довільному порядку, оскільки ці файли потрібно завантажити з сервера. Ті самі правила застосовуються до завантаження ігрових елементів, таких як спрайти та звуки. Це метод, який ви використовували дотепер для завантаження ігрових складових:

```
var sprite = new Image();
sprite.src = "someImageFile.png";
var anotherSprite = new Image();
anotherSprite.src = "anotherImageFile.png";
// and so on
```

Це видається простим. Для кожного спрайту, який необхідно завантажити, ви створюєте об'єкт Image і призначаєте значення його змінній src. Присвоювання змінній src певного значення не означає, що зображення відразу ж завантажиться. Воно просто говорить браузеру, щоб почати отримання цього зображення з сервера. Залежно від швидкості підключення до Інтернету, це може зайняти деякий час. Якщо ви спробуєте намалювати зображення занадто рано, браузер зупинить сценарій через помилку доступу (намагається намалювати зображення, яке ще не завантажено). Для того, щоб уникнути цієї проблеми, в попередньому прикладі було завантажено спрайт таким чином:

```
sprites.background = new Image();
sprites.background.src = spriteFolder + "spr_background.jpg";
sprites.cannon_barrel = new Image();
sprites.cannon_barrel.src = spriteFolder + "spr_cannon_barrel.png";
sprites.cannon_red = new Image();
sprites.cannon_red.src = spriteFolder + "spr_cannon_red.png";
sprites.cannon_green = new Image();
sprites.cannon_green.src = spriteFolder + "spr_cannon_green.png";
sprites.cannon_blue = new Image();
sprites.cannon_blue.src = spriteFolder + "spr_cannon_blue.png";
```

```
cannon.initialize();
window.setTimeout(Game.mainLoop, 500);
```

Зверніть увагу на останній рядок коду жирним шрифтом. Після налаштування змінних src всіх об'єктів Image ви повідомляєте браузеру, що він чекає 500 мілісекунд, перш ніж виконувати основний цикл.Таким чином, браузер повинен мати достатньо часу для завантаження спрайтів. Але що робити, якщо підключення до Інтернету є занадто повільним? Тоді може бути недостатньо 500 мілісекунд. Або що, якщо підключення до Інтернету дуже швидке? Тоді ви дозволяєте гравцю чекати даремно. Для того, щоб вирішити цю проблему, вам потрібно, щоб програма зачекала, поки всі зображення буде завантажено перед виконанням основного циклу. Ви побачите, як це зробити правильно з функціями обробника подій. Але до цього давайте поговоримо трохи про методи та функції.

4.3 Методи та Функції

Ви вже бачили і використовували множину різних методів і функцій. Наприклад, існує чітка різниця між методом Canvas2D.drawImage та методом cannon.update: останній не має жодних параметрів, тоді як перший має (спрайт, його положення, його обертання та його джерело). Крім того, деякі функції/методи можуть мати результуюче значення об'єкта, яке можна використовувати в інструкції, яка викликає метод, наприклад, зберігаючи результат у змінній:

```
var n = Math.random ();
```

Тут ви викликаете функцію random, яка визначена як а частина об'єкту Math і зберігаєте результат у змінній n. random забезпечує значення результату, яке може бути збережене. Метод Canvas2D.drawImage з іншого боку, не дає результату, який можна зберегти у змінній. Звичайно, метод має певний ефект, тому що він малює спрайт на екрані, який також можна розглядати в якості результату виклику методу. Проте, коли ми говоримо про результат методу, це означає, що метод повертае значення, яке може бути збережене в змінній. Це також називається зворотним значенням методу або функції. У математиці, як правило, функція має результат. Математична функція $f(x) = x^2$ приймає як параметр значення x і повертає його квадрат в результаті. Ви можете написати цю математичну функцію в JavaScript:

```
var square = function (x) {
return x * x;
}
```

Якщо ви подивитесь на заголовок цього методу, ви побачите, що він

приймає один параметр з назвою x . Оскільки функція повертає значення, ви можете зберегти це значення в змінній:

```
var sx = square(10);
```

Після виконання цієї інструкції змінна sx буде містити значення 100. У тілі функції ви можете вказати, використовуючи ключове слово return, що є фактичним значенням функції.

Прикладом методу, який не повертає значення, є cannon.handleInput. Оскільки цей метод не має повернутого значення, вам не потрібно використовувати ключове слово return в тілі методу, хоча це іноді може бути корисним. Наприклад, припустимо, що необхідно змінити колір гармати, якщо миша знаходиться в лівій частині екрана. Ви можете досягти цього наступним чином:

```
cannon.handleInput = function () {
if (Mouse.position.x > 10)
return;
if (Keyboard.keyDown === Keys.R)
cannon.currentColor = sprites.cannon_red;
else if (Keyboard.keyDown === Keys.G)
// etc.
};
```

У цьому методі ви спочатку перевіряєте, чи x-позиція миші перевищує 10. Якщо це так, ви виконуєте інструкцію return. Після цього будь-які інструкції більше не будуть виконуватися.

Зверніть увагу, що кожного разу, коли викликається метод без зворотного значення, він не має результату, який можна зберегти в змінній.

Наприклад:

```
var what = cannon.handleInput ();
```

Якщо метод або функція повертає значення, це значення не обов'язково потрібно зберігати в змінній. Ви також можете безпосередньо використовувати його в if інструкції, як і в cannon.handleInput :

```
if (Math.random() > 0.5)
// do something
```

Завантаження компонентів ігри правильним шляхом Для того, щоб зробити завантаження спрайтів трохи простішим, додамо метод loadSprite до об'єкта Game :

```
Game.loadSprite = function(imageName) {
var image = new Image();
image.src = imageName; return image;
}
```

Код для завантаження різних спрайтів тепер стає набагато коротшим:

```
var sprFolder = "../../assets/Painter/sprites/ ";
sprites.background = Game.loadSprite (sprFolder + "spr_background.jpg");
sprites.cannon_barrel = Game.loadSprite (sprFolder + "spr_cannon_barrel.png");
sprites.cannon_red = Game.loadSprite (sprFolder + "spr_cannon_red.png");
sprites.cannon_green = Game.loadSprite (sprFolder + "spr_cannon_green.png");
sprites.cannon_blue = Game.loadSprite (sprFolder + "spr_cannon_blue.png");
```

Проте проблема вирішення часу, необхідного для завантаження спрацьовування, ще не вирішена. Щоб вирішити цю проблему, перше, що вам потрібно зробити, це відстежувати кількість спрайтів, які ви завантажуєте. Ви це робите, додавши змінну до об'єкта Game, що називається spritesStillLoading :

```
var Game = {
spritesStillLoading : 0
};
```

Спочатку для цієї змінної встановлено значення 0. Кожного разу, коли ви завантажуєте спрайт, ви збільшуєте змінну на 1. Ви це робите в методі loadSprite :

```
Game.loadSprite = function(imageName) {
var image = new Image();
image.src = imageName; Game.spritesStillLoading += 1; return image;
}
```

Отже, тепер, кожного разу, коли ви завантажуєте спрайт, значення змінної spritesStillLoading збільшується. Далі ви хочете зменшити цю змінну кожного разу, коли спрайт завершив завантаження. Це можна зробити, використовуючи функцію обробника подій. Ви призначаєте цю функцію для змінної onload в об'єкті image.У тілі цієї функції ви зменшуєте змінну. Ось версія методу loadSprite, який додає обробник події:

```
Game.loadSprite = function (imageName) {
var image = new Image();
image.src = imageName; Game.spritesStillLoading += 1;
image.onload = function () {
Game.spritesStillLoading -= 1;
```

```
};
return image;
};
```

Тепер змінна spritesStillLoading точно відображає, скільки спрайтів ще потрібно завантажити. Ви можете використовувати цю інформацію, щоб почекати, коли запустити основний цикл, доки ця змінна не містить значення 0. Для цього ви створюєте два методи з циклами: цикл завантаження активності та основний цикл гри. У циклі завантаження спрайту ви просто перевіряєте, чи є ще файли, які потрібно завантажити.

Якщо це так, ви знову визиваєте цикл завантаження спрайтів. Якщо всі спрайти були завантажені, ви називаєте метод основного циклу. Ось метод циклу завантаження спрайтів:

```
Game.assetLoadingLoop = function () {
if (Game.spritesStillLoading > 0)
window.setTimeout(Game.assetLoadingLoop, 1000 / 60);
else {
Game.initialize();
Game.mainLoop();
}
};
```

4.4 Написання більш ефективного циклу гри

До цього часу ви використовували метод window.setTimeout для створення циклу гри. Проблема в тому, що не всі веб-браузери та їх версії використовують одне ім'я методу. Новіші версії найбільш популярних веб-браузерів використовують метод window.requestAnimationFrame. Проте старіші версії Firefox використовують window.mozRequestAnimationFrame, а старіші версії Safari та Chrome використовують window.webkitRequestAnimationFrame. Оскільки більшість веб-браузерів вже використовують метод window.requestAnimationFrame, ви можете розширити визначення цього методу таким чином:

```
window.requestAnimationFrame = window.requestAnimationFrame ||
window.webkitRequestAnimationFrame ||
window.mozRequestAnimationFrame ||
window.oRequestAnimationFrame || window.msRequestAnimationFrame ||
function (callback) {
window.setTimeout(callback, 1000 / 60);
};
```

Оператор || тут використовується, щоб визначити, на який метод ссилається

window.requestAnimationFrame. Якщо перший параметр не визначено (наприклад, якщо ви маєте справу зі старим веб-браузером), ви перевіряєте будь-які старіші імена. Якщо жоден з оптимізованих методів ігрового циклу не доступний, ви дозволяєте requestAnimationFrame вказувати на функцію, яка викликає метод window.setTimeout. У JavaScript змінна window є контейнером глобального простору імен. Це означає, що ви можете викликати метод requestAnimationFrame з або без змінної window перед ним.

Виклик:

```
window.requestAnimationFrame (callbackFunction);
```

еквівалентно

```
requestAnimationFrame (callbackFunction);
```

У прикладі Painter3 використовується оптимізований метод циклу гри:

```
Game.assetLoadingLoop = function () {
if (Game.spritesStillLoading > 0)
window.requestAnimationFrame(Game.assetLoadingLoop);
else {
Game.initialize();
Game.mainLoop();
}
};
```

І аналогічно, ось метод mainLoop у об'єкті Game:

```
Game.mainLoop = function () {
Game.handleInput();
Game.update();
Game.draw();
Mouse.reset();
window.requestAnimationFrame(Game.mainLoop);
};
```

4.5 Розділення загального коду від коду специфічного для гри

Раніше ми не робили жодного розмежування між кодом, який можна використовувати для багатьох різних ігор та коду, характерних для однієї гри. Деякі частини написаного нами коду, наприклад, метод Game.mainLoop, також будуть корисними для інших ігор JavaScript. Те саме стосується і всього коду, який ви написали для завантаження спрайтів. Тепер, коли ви бачили спосіб розділити код на різні файли сценаріїв, ви можете використовувати це для

вашої переваги. Відокремлюючи загальний код від коду, специфічного для Painter, його буде простіше повторно використовувати. Якщо ви хочете повторно використовувати код завантаження спрайтів, ви просто повинні включити вихідний файл, що містить код, у свою нову ігрову програму.

Приклад Painter4 створює окремий файл Game.js, який містить об'єкт Game та ряд корисних методів, що належать цьому об'єкту. Частини, специфічні для Painter, були перенесені в файл Painter.js .У цьому файлі є спосіб завантаження спрайтів, а також метод для ініціалізації гри. Крім того, новий файл під назвою PainterGameWorld.js обробляє різні об'єкти в грі. У попередніх версіях Painter цей ігровий світ складався лише з фонового зображення та гармати. У наступному розділі ви додаєте м'яч до цього ігровому світу. Тоді ігровий світ гри Painter визначається об'єктом, який гарантує, що всі об'єкти гри оновлюються та малюються. Це частина визначення обєкту painterGameWorld:

```
var painterGameWorld = {
};
painterGameWorld.handleInput = function (delta) {
ball.handleInput(delta);
cannon.handleInput(delta);
};
painterGameWorld.update = function (delta) {
ball.update(delta);
cannon.update(delta);
};
painterGameWorld.draw = function () {
Canvas2D.drawImage(sprites.background, { x : 0, y : 0 }, 0,
{ x : 0, y : 0 });
ball.draw();
cannon.draw();
};
```

Під час ініціалізації гри ви ініціалізуєте об'єкти гри, і повідомляєте об'єкту Game, що об'єкт, що керує ігровим світом, - painterGameWorld , таким чином:

```
Game.initialize = function () {
cannon.initialize();
ball.initialize();
Game.gameWorld = painterGameWorld;
};
```

Всередині методу Game.mainLoop тепер потрібно лише викликати правильні методи:

```
Game.mainLoop = function () {
Game.gameWorld.handleInput();
Game.gameWorld.update();
Canvas2D.clear();
```

```
Game.gameWorld.draw();
Mouse.reset();
requestAnimationFrame(Game.mainLoop);
};
```

Як результат, ви добре відокремили загальний код гри (у Game.js) та код гри, який полягає в завантаженні спрайтів та ініціалізації гри (Painter.js) та оновлення та малювання об'єктів гри Painter (PainterGameWorld .js). Будь-які інші об'єкти гри, специфічні для Painter, визначаються у власному файлі сценарію (наприклад, Cannon.js).

Лекція 5
Структурування програмного коду ігрового веб-додатка мовою програмування JavaScript за допомогою об'єктів та методів.

Анотація. Лекція знайомить з прийомами структурування програмного кода з використанням об'єктів та за допомогою розбиття кода на окремі файли.

Мета лекції:

- Ознайомити студентів з прийомами опису об'єктів в окремих файлах.
- Ознайомити студентів зі способом відокремлення загального коду та ігрового коду.

5.1 Створення декількох об'єктів однакового типу

До цього часу у Painter було лише по одному виду кожного об'єкта гри. Є тільки одна гармата і один м'яч. Те саме стосується всіх інших об'єктів у коді JavaScript. Є об'єкт Game, один об'єкт Keyboard, один об'єкт Mouse і так далі. Ви створюєте ці об'єкти, оголосивши змінну, що відноситься до порожнього або складного об'єкта, і додає до неї корисні методи. Наприклад, ось як ви створюєте об'єкт ball:

```
var ball = {
};
ball.initialize = function() {
ball.position = { x : 0, y : 0 };
// etc.
};
ball.handleInput = function (delta) {
if (Mouse.leftPressed && !ball.shooting) {
```

Довідник модуля
«Розробка ігрових веб-додатків»
40 | 48

```
// do something
}
};
// etc.
```

Припустимо, ви хочете мати можливість стріляти трьома м'ячами одночасно в грі Painter. Якщо робити це так, як ви створювали об'єкти до цих пір, ви створите дві змінні, ball2 і ball3, і двічі скопіюйте код, який ви використовували для об'єкта ball. Це не дуже хороше рішення з кількох причин. Для одного коду копіювання означає, що ви повинні мати справу з проблемами керування версією. Наприклад, що, якщо ви знайдете помилку в коді методу update? Ви повинні переконатися, що ви копіюєте покращений код до інших об'єктів ball. Якщо ви забули одну копію, помилка залишається там, коли ви вважаєте, що ви вирішили це. Інша проблема полягає в тому, що цей підхід не дуже добре розвивається. Що станеться, якщо ви захочете розширити гру так, щоб гравець міг стріляти 20 м'ячами одночасно? Ви будете копіювати код 20 разів? Також зауважте, що чим більше файли JavaScript, тим більше триває завантаження та інтерпретація браузера. Так що, якщо ви не хочете, щоб ваші гравці чекати надто довго для сценаріїв завантаження, то краще уникнути копіювання коду. Нарешті, дубльований код виглядає потворно, зачіпає файли вихідного коду, а також ускладнює пошук інших розділів потрібного вам коду, що призводить до надмірної прокрутки та загальному зменшенні кодуванні ефективність На щастя, є дуже хороше рішення цієї проблеми. Це конструктор JavaScript для програмування, званий *прототипом*. Прототипи включають змінні та методи, які містить об’єкт. Після визначення прототипу ви можете створювати об'єкти, використовуючи цей прототип, одним рядком коду!

Подивіться на цю строку коду:

```
var image = new Image();
```

Тут ви створюєте об'єкт image, який використовує прототип Image для самостійного побудови. Визначити прототип легко. Погляньте на цей приклад:

```
function Dog() {
}
Dog.prototype.bark = function () {
console.log("woof!");
};
```

Це створює функцію, яка називається Dog. Коли ця функція викликається у поєднанні з ключовим словом new, об'єкт створюється. Кожна функція в

JavaScript має *прототип,* який містить інформацію про те, як об'єкти повинні виглядат. Цей приклад визначає метод, названий bark, що є частиною прототипу Dog. Ось як можна створили новий об'єкт Dog:

```
var lucy = new Dog();
```

Оскільки lucy створюється відповідно до прототипу функції Dog , об'єкт lucy містить метод, названий bark:

```
lucy.bark(); // outputs "woof!" to the console
```

Приємно, що тепер ви можете створити багато собак, які можуть гавкати, але вам потрібно визначити метод bark один раз:

```
var max = new Dog();
var zoe = new Dog();
var buster = new Dog();
max.bark();
zoe.bark(); b
uster.bark();
```

Тепер ви можете створити стільки куль, скільки хочете, і ініціалізувати їх:

```
var ball = new Ball();
var anotherBall = new Ball();
ball.initialize();
anotherBall.initialize();
```

Кожного разу, коли ви створюєте новий м'яч, до об'єкту додаються будь-які методи з прототипа. Коли метод initialize викликається для об'єкта ball, this відноситься до ball. Коли його називають anotherBall , this відноситься до anotherBall.

Ви можете фактично скоротити код, який ви написали. Навіщо додавати метод initialize, коли сам Ball вже є функцією, яка викликається? Ви можете просто виконати ініціалізацію в цій функції, як показано нижче:

```
function Ball() {
this.position = { x : 0, y : 0 };
this.velocity = { x : 0, y : 0 };
this.origin = { x : 0, y : 0 };
this.currentColor = sprites.ball_red;
this.shooting = false;
}
```

Тепер, коли ви створюєте кулі, вони ініціалізуються при створенні:

```
var ball = new Ball();
var anotherBall = new Ball();
```

А оскільки Ball є функцією, ви можете навіть передати параметри, якщо хочете:

```
function Ball(pos) {
this.position = pos;
this.velocity = { x : 0, y : 0 };
this.origin = { x : 0, y : 0 };
this.currentColor = sprites.ball_red;
this.shooting = false;
}
var ball = new Ball({ x : 0, y : 0});
var anotherBall = new Ball({ x : 100, y : 100});
```

Оскільки функція Ball відповідає за ініціалізацію (або *конструювання*) об'єкта, ця функція також називається *конструктором*. Конструктор разом із методами, визначеними прототипом, називається *класом*. Коли об'єкт створюється відповідно до класу, ви також кажете, що об'єкт має цей клас як *тип*. У попередньому прикладі об'єкт ball має тип Ball, оскільки він був створений за допомогою конструктора Ball та його прототипу. Клас є *орієнтиром* для об'єкта, і таким чином він описує дві речі:

Дані, що містяться в об'єкті. У випадку з кулями ці дані складаються з позиції, швидкості, початку відліку, поточного кольору та змінної, яка вказує на те, чи в повітрі мяч. Як правило, ці дані ініціалізуються в конструкторі.

Методи *маніпулювання* даними. У класі Ball ці методи є методами циклічної гри (handleInput , update , draw і reset).

Ви можете дуже легко перевести методи ігрового циклу як методи в прототипі Ball, просто замінивши ball на this. Наприклад, метод handleInput:

```
Ball.prototype.handleInput = function (delta) {
if (Mouse.leftPressed && !this.shooting) {
this.shooting = true;
this.velocity.x = (Mouse.position.x - this.position.x) * 1.2;
this.velocity.y = (Mouse.position.y - this.position.y) * 1.2;
}
};
```

Поняття класів та об'єктів надзвичайно потужне. Воно ґрунтується на *парадигмі об'єктно-орієнтованого програмування*. JavaScript є дуже гнучкою мовою, оскільки вона не зобов'язує вас використовувати класи. Ви могли б писати скрипти, використовуючи лише функції, якщо хочете (і це те, що ви

зробили дотепер). Але оскільки класи є такими потужними концепціями програмування і широко використовуються в (ігровій) галузі, навчившись належним чином використовувати класи, ви можете спроектувати набагато краще програмне забезпечення в *будь-якій* мові програмування.

5.2 **Побудова ігрових об'єктів, частин ігрового світу**

Тепер, коли ви бачили, як створювати класи, вам потрібно переосмислити, де побудувати об'єкти гри. До цих пір об'єкти гри були оголошені глобальними змінними, і таким чином вони були доступні скрізь. Наприклад, це те, як ви створюєте об'єкт cannon:

```
var cannon = {
};
cannon.initialize = function() {
cannon.position = { x : 72, y : 405 };
cannon.colorPosition = { x : 55, y : 388 };
cannon.origin = { x : 34, y : 34 };
cannon.currentColor = sprites.cannon_red;
cannon.rotation = 0;
};
```

У прикладі Painter5 це конструктор класу Cannon виглядає так:

```
function Cannon() {
this.position = { x : 72, y : 405 };
this.colorPosition = { x : 55, y : 388 };
this.origin = { x : 34, y : 34 };
this.currentColor = sprites.cannon_red;
this.rotation = 0;
}
```

У методі update м'яча потрібно отримати поточний колір гармати, щоб ви могли оновити колір м'яча. Ось як ви робили це раніше:

```
if (cannon.currentColor === sprites.cannon_red)
ball.currentColor = sprites.ball_red;
elseif (cannon.currentColor === sprites.cannon_green)
ball.currentColor = sprites.ball_green;
else
ball.currentColor = sprites.ball_blue;
```

При визначенні класу з використанням прототипного підходу JavaScript, ви повинні замінити ball на this (оскільки немає іменованого екземпляра об'єкта).

Тому попередній код перекладено так:

```
if (cannon.currentColor === sprites.cannon_red)
this.currentColor = sprites.ball_red;
elseif (cannon.currentColor === sprites.cannon_green)
this.currentColor = sprites.ball_green;
else
this.currentColor = sprites.ball_blue;
```

Але як звертатись до об'єкта cannon, якщо гармати також побудовані з використанням класу?Це ставить два запитання:
Де в коді будувати ігрові об'єкти?
Як посилатись на ці об'єкти гри, якщо вони не є глобальними змінними?

Логічно, що об'єкти гри повинні бути побудовані, коли побудований ігровий світ. Ось чому приклад Painter5 створює об'єкти гри в класі PainterGameWorld (який раніше був об'єктом painterGameWorld). Ось частина конструктора цього класу:

```
function PainterGameWorld() {
this.cannon = new Cannon();
this.ball = new Ball();
// create more game objects if needed
}
```

Отже, це відповідає першому питанню, але це ставить ще одне питання. Якщо ігрові об'єкти створюються під час створення ігрового світу, де викликати конструктор PainterGameWorld для створення ігрового світу?

Якщо ви відкриєте файл Game.js , ви побачите, що існує ще один клас, визначений за методом прототипу: Game_Singleton . Це його конструктор:

```
function Game_Singleton() {
this.size = undefined;
this.spritesStillLoading = 0;
this.gameWorld = undefined;
}
```

Як ви можете бачити, цей клас може будувати об'єкт Game, який був використаний в попередньому розділі. Клас Game_Singleton має метод Initialize, в якому створюється ігровий світ об'єкта:

```
Game_Singleton.prototype.initialize = function () {
this.gameWorld = new PainterGameWorld();
};
```

Добре, ви виявили, де побудований ігровий світ. Але де ж побудовано екземпляр об'єкта Game_Singleton? Вам потрібен цей екземпляр для доступу до ігрового світу, який, у свою чергу, дасть вам доступ до об'єктів гри. Якщо ви подивитеся на останній рядок файлу Game.js , ви побачите цю інструкцію:

```
var Game = new Game_Singleton ();
```

Нарешті, фактичне оголошення змінної! Тому за допомогою змінної Game ви можете отримати доступ до ігрового світу; і через цей об'єкт ви можете отримати доступ до ігрових об'єктів, які є частиною ігрового світу.

Наприклад, це те, як ви отримуєте об'єкт cannon:

```
Game.gameWorld.cannon
```

Чому б не просто оголосити кожен об'єкт гри глобальною змінною, як раніше? Існує кілька причин. По-перше, оголошуючи багато глобальних змінних в різних місцях, ваш код стає складнішим для повторного використання. Припустимо, ви хочете використовувати частину коду від Painter в іншій програмі, яка також використовує кулі та гармати. Тепер ви повинні просіяти код, щоб знайти, де були оголошені глобальні змінні, і переконайтеся, що вони корисні для вашої програми. Набагато краще оголосити ці змінні в одному місці (наприклад, клас PainterGameWorld), тому ці оголошення легше знайти.

Другою складністю із використанням багатьох глобальних змінних є те, що ви відкидаєте будь-яку структуру або відносини, які існують між змінами. У грі Painter зрозуміло, що гармата та м'яч *є частиною ігрового світу* .Ваш код стає легше зрозуміти, якщо ви прямо висловите це відношення, дозволяючи об'єктам гри входити в об'єкт ігрового світу.

Використовуючи нову структуру, в якій об'єкт Game є деревом інших об'єктів, тепер ви можете отримати доступ до об'єкта cannon, щоб отримати бажаний колір кулі, таким чином:

```
if (Game.gameWorld.cannon.currentColor === sprites.cannon_red)
this.currentColor = sprites.ball_red;
elseif (Game.gameWorld.cannon.currentColor === sprites.cannon_green)
this.currentColor = sprites.ball_green;
else
this.currentColor = sprites.ball_blue;
```

Іноді може бути корисним намалювати дерево-структуру ігрових об'єктів на папері або створити діаграму, де пізніше можна поставити посилання з власними іменами. Оскільки ігри, які ви розробляєте, стають все більш складними, таке дерево забезпечує корисний огляд того, який об'єкт належить

чому, і це заощаджує вас від необхідності повторно створювати це дерево психічно під час роботи з кодом.

5.2 Написання класу з кількома екземплярами

Тепер, коли ви можете побудувати кілька об'єктів одного і того ж типу, давайте додамо кілька банок для фарби до гри Painter. Ці фарби повинні бути задані випадковим кольором, і вони повинні падати з верхньої частини екрану. Коли вони вийшли з нижньої частини екрана, ви призначаєте їм новий колір і переміщаєте їх у верхній кут. Для гравця, здається, ніби різноманітні фарби кожного разу падають. Насправді, вам потрібно лише три об'єкти, які можуть бути використані повторно. У класі PaintCan ви визначаєте можливі фарби і їх поведінку. Потім ви можете створити кілька екземплярів цього класу. У класі PainterGameWorld ви зберігаєте ці екземпляри у трьох різних змінних членах, які оголошуються та ініціалізуються у конструкторі PainterGameWorld :

```
function PainterGameWorld() {
this.cannon = new Cannon();
this.ball = new Ball();
this.can1 = new PaintCan(450);
this.can2 = new PaintCan(575);
this.can3 = new PaintCan(700);
}
```

Різниця між класом PaintCan і класами Ball і Cannon полягає в тому, що фарби для банок мають різні позиції. Ось чому ви передаєте значення координат як параметр, коли конструюються контейнери фарби. Це значення вказує на бажане положення x фарби. Позиція y не повинна бути надана, тому що вона буде розрахована на основі y-швидкості кожної фарби. Щоб зробити речі цікавішими, ви дозволяєте банкам падати з різними, випадковими швидкостями.

Конструктор класу PaintCan :

```
function PaintCan(xPosition) {
this.currentColor = sprites.can_red;
this.velocity = new Vector2();
this.position = new Vector2(xPosition, -200);
this.origin = new Vector2();
this.reset();
}
```

Точно так само, як гармата та м'яч, фарба може мати певний колір. За замовчуванням ви обираєте спрайт червоної фарбиа. Спочатку ви встановлюєте положення y фарби таким чином, щоб вона була намальована безпосередньо

поза вершиною екрана, так що пізніше в грі ви можете побачити її падіння. У конструкторі PainterGameWorld тричі ви визиваєте цей конструктор, щоб створити три об'єкти PaintCan, кожен з яких має різні позиції x.

Оскільки банки з фарбою не обробляють ніяких вхідних даних (тільки куля та гармата роблять це), то вам не потрібен метод handleInput для цього класу. Проте банки потрібно оновлювати. Одна з речей, які ви хочете зробити, полягає в тому, щоб фарби падали у випадкових місцях і з довільною швидкістю. Але як ви можете це зробити?

5.4 Випадковість в грі

Одна з найважливіших частин поведінки фарби - це те, що деякі її аспекти повинні бути *непередбачуваними*. Ви не хочете, щоб кожна банка падала з передбачуваною швидкістю або випадковим часом. Ви хочете додати фактор *випадковості,* так що кожного разу, коли гравець починає нову гру, гра буде іншою. Звичайно, вам також потрібно зберегти цю випадковість під контролем. Швидкість повинна бути випадковою, але в межах *відтворюваного діапазону швидкості.*

Що означає випадковість насправді? Як правило, випадкові події або значення в іграх та інших програмах управляються генератором *випадкових чисел.* У JavaScript існує метод random який є частиною об'єкта Math. Ви можете задатися питанням: як комп'ютер створює цілком випадкове число?

Чи дійсно існує випадковість? Чи випадковість є лише проявом поведінки, яку ви ще не можете повністю прогнозувати, і тому називаєте "випадковістю"? У ігрових світах та комп'ютерних програмах ви *можете* точно передбачити, що станеться, оскільки комп'ютер може виконувати лише те, що ви запрограмуєте. Тому, строго кажучи, комп'ютер не здатний створювати цілком випадкове число. Спосіб створення випадкових чисел називається *генератором псевдовипадкових чисел.* Більшість генераторів випадкових чисел можуть генерувати число в діапазоні, наприклад, від 0 до 1, але вони також можуть генерувати довільне число або число в іншому діапазоні. Кожен номер у межах діапазону має рівні шанси на створення. У статистиці такий розподіл називається *рівномірним розподілом* .

Як ви використовуєте генератор випадкових чисел для створення випадковості у вашому ігровому світі? Припустимо, ви хочете створити ворога в 75% випадків, коли користувач проходить через двері. У цьому випадку ви створюєте випадкове число від 0 до 1. Якщо число менше або дорівнює 0,75, ви породжуєте ворога; інакше ні. Через рівномірний розподіл це призведе саме до поведінки, яка вам потрібна. Наступний код JavaScript показує це:

```
var spawnEnemyProbability = Math.random();
if (spawnEnemyProbability >=0.75)
// spawn an enemy
else
```

```
// do something else
```

Якщо ви хочете розрахувати випадкову швидкість від 0,5 до 1, ви генеруєте випадкове число від 0 до 1, поділіть це число на 2 і додайте 0,5:

```
var newSpeed = Math.random () / 2 * 0.5;__
```

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Вступ

Комплекс лабораторних робіт модуля «Розробка ігрових web-додактків» виконується відповідно до програми навчальної дисципліни «Розробка ігрових web-додактків» та передбачає поглиблення теоретичних знань з навчальної дисципліни та набуття практичних навичок роботи з JavaScript за допомогою спеціалізованого ігрового обладнання.

До комплексу лабораторних робіт входять індивідуальні завдання, які передбачають проектування та розробку власного ігрового web-додатку. Тема індивідуального завдання вибирається студентом самостійно. Індивідуальні завдання виконуються студентом самостійно з консультацією викладача.

Максимальна сумарна оцінка за виконання лабораторних робіт в межах навчальної дисципліни складає 6 балів. У кожній роботі 30% нараховується за виконання практичної частини і її відповідність поставленій задачі і 70% – за захист роботи (аргументоване пояснення ходу роботи і відповіді на теоретичні питання).

Лабораторна робота 1
Створення ігрового світу.

Анотація. Лабораторна робота орієнтована на ознайомлення студентів з основами побудови ігрового світу мовою програмування JavaScript.

Мета лабораторної роботи:

- Ознайомитись з основними типами змінних, а також об’єктами, які містять змінні-члени та методи.
- Ознайомитись з декларацією та ініціалізацією змінних, з поняттям глобальні змінні, операторами мови JavaScript.
- Навчитися створювати простий веб-додаток, в якому геометрична фігура переміщується по полотну.

У разі успішного виконання лабораторної роботи студент буде знати, як зберігати інформацію в змінних, як створювати об’єкти, які містять змінні-члени та методи, як використовувати метод update для зміни ігрового світу та метод draw для відображення ігрового світу на екрані.

1.1 Основні типи даних і змінні

Типи даних

Типи даних представляють різні види структурованої інформації. Браузер/інтерпретатор може розрізняти всі ці різні види інформації, і в багатьох

випадках навіть конвертувати інформацію одного типу в інший.

Наприклад, в JavaScript ви можете використовувати як окремі, так і подвійні лапки для представлення тексту. Наприклад, наступні два інструкції роблять одне й те ж саме:

```
canvas = document.getElementById ("myCanvas");
canvas = document.getElementById ('myCanvas');
```

Браузери можуть автоматично конвертувати різні типи інформації. Наприклад, наступне не призведе до синтаксичної помилки:

```
canvas = document.getElementById (12);
```

Номер, переданий як параметр, просто перетворюється в текст. У цьому випадку, звичайно, немає полотна з ідентифікатором 12, тому програма більше правильною. Але якщо ви хочете замінити ідентифікатор

полотна таким чином, програма буде працювати правильно:

```
<canvas id = "12" width = "800" height = "480"> </ canvas>
```

Браузер автоматично перетворює текст і цифри. Більшість мов програмування значно суворіші, ніж JavaScript. У таких мовах, як Java та C #, конверсія між типами виконується дуже обмежено. У більшості випадків ви повинні явно сказати компілятору, що необхідно здійснити конверсію між типами.

Яка причина для більш жорсткої політики щодо перетворення типу? З одного боку, чітко визначаючи, який тип функції або метод очікує як параметр, іншим програмістам легше зрозуміти, як використовувати цю функцію. Подивіться на наступний заголовок, наприклад:

```
function playAudio (audioFileId)
```

Тільки дивлячись на цей заголовок, ви не можете бути впевнені, audioFileId – це число чи текст. У C # заголовок подібного методу виглядає так:

```
void playAudio (string audioFileId)
```

ви можете побачити. що заголовок має не тільки ім’я, але й *тип*, що належить до імені.

1.2 Декларація і присвоювання значень змінним

В JavaScript дуже просто можно зберігати інформацію і використовувати її пізніше. Вче, що потрібно зробити, це вказати назву, яку ви використовуєте,

коли звертаєтесь до цієї інформації. Це ім'я називається *змінною*. Якщо ви хочете використовувати змінну в вашій програмі, можна декларувати її, перш ніж ви на насправді будете її використовувати. Ось, як декларуеється змінна:

```
var red;
```

У цьому прикладі red є назвою змінної. Ви можете використовувати змінну у вашій програмі для зберігання інформації, яка вам знадобиться пізніше.

Коли ви оголошуєте змінну, вам не потрібно вказувати тип інформації, яку ви зберігаєте.

Зінна – це просто місце в пам'яті, що має назву. Дуже небагато мов програмування вимагає вказування типу змінної під час декларування.

Наприклад, це стосується таких мов, як C ++ або Java. Однак багато мов сценаріїв (включаючи JavaScript) дозволяють оголосити змінну без визначення її типу. У JavaScript ви можете одночасно оголосити більше однієї змінної. Наприклад:

```
var red, green, fridge, grandMa, applePie;
```

Тут ви оголошуєте п'ять різних змінних, які ви можете використовувати у вашій програмі. Коли ви оголошуєте ці змінні, вони ще не містять значення. У цьому випадку ці змінні вважаються *невизначеними*. Ви можете призначити значення для змінної, використовуючи *команду присвоєння*. Наприклад, давайте присвоїмо значення змінній red наступним чином:

```
red = 3;
```

Інструкція присвоєння складається з наступних частин:

- назва змінної
- знак “=”
- нове значення змінної
- крапка з комою

Ви можете визначити вказівку присвоєння знаком =. Однак, краще думати про цей знак, як "стає", а не "є рівним" у JavaScript.

Синтаксична діаграма, що описує інструкцію присвоєння, наведена на рисунку 1.1.

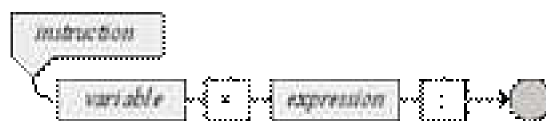


Рис.1.1 *Синтаксична схема інструкції присвоєння*

Отже, тепер ви побачили одну інструкцію для оголошення змінної та іншу інструкцію для зберігання в ній значення. Але якщо ви вже знаєте, яке значення ви хочете зберегти в змінній при оголошенні, ви можете поєднати декларацію змінної та перше призначення:

```
var red = 3;
```

Коли ця інструкція буде виконана, пам'ять буде містити значення 3, як показано на малюнку 1.2 .

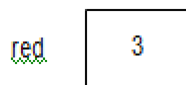


Рис.1.2 *Пам'ять після декларації та присвоєння змінної*

Ось кілька прикладів більшої кількості декларацій та призначень числових змінних:

```
var age = 16;
var numberOfBananas;
numberOfBananas = 2;
var a, b;
a = 4;
var c = 4, d = 15, e = -3; c = d;
numberOfBananas = age + 12;
```

У четвертому рядку цього прикладу, ви бачите, що можна декларувати кілька змінних в одному виразі. Ви навіть можете виконувати декілька декларувань з присвоюваннями в одній декларації, як можна бачити в шостому рядку коду прикладу. На правій стороні присвоювання ви можете поставити інші змінні або математичні вирази, як ви можете побачити в двох останніх рядках. Інструкція c = d; призводить до значення, збереженого в змінній d, яке таклж зберігається в змінній c. Так як змінна d містить значення 15, після того, як ця команда виконується, змінна c також містить значення 15. Остання інструкція приймає значення, що зберігається в змінній age (16), додає 12 до нього і зберігає результат у змінній numberOfBananas (який зараз має значення 28). Таким чином, пам'ять виглядає так, як показано на малюнку 1.3, після виконання цих інструкцій.

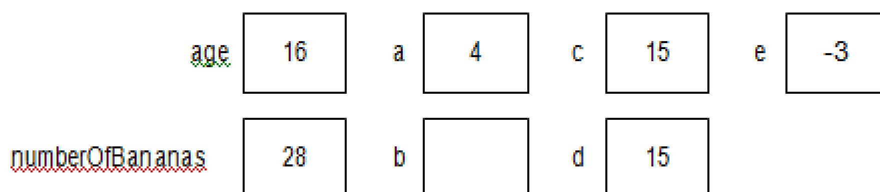


Рис.1.3 *Огляд пам'яті після декларування та призначення декількох змінних*

Синтаксис декларування змінних (з необов'язковою ініціалізацією) виражається на діаграмі, показаній на рисунку 1.4 .

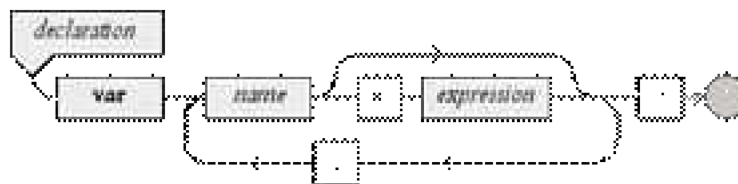


Рис.1.4 *Синтаксична діаграма декларації змінної з необов'язковою ініціалізацією*

1.2 Глобальні змінні та строгий режим

Замість того, щоб оголосити змінну перед використанням, в JavaScript також можливо просто почати використовувати змінну без декларування.

Наприклад, розглянемо наступні інструкції:

```
var a = 3;
var b;
b = 4;
x = a + b;
```

Як ви можете бачити, змінні a і b оголошуються в перших двох інструкціях, використовуючи ключове слово var. Змінна x ніколи не оголошується, але вона використовується для збереження суми двох змінних.

JavaScript дозволяє це зробити. Проте це дуже погана практика, і ось чому. Проблема простого використання змінної без декларування полягає в тому, що інтерпретатор JavaScript автоматично декларує цю змінну для вас, без вашого усвідомлення. Якщо ви випадково використовуєте змінну з тим самим іменем де-небудь ще, ваша програма може відображати поведінку, яку ви не очікуєте, оскільки ця змінна вже існує. Крім того, якщо ви використовуєте множину різних змінних, вам слід відстежувати ці глобальні змінні. Але ще більша проблема виявляється в наступному прикладі:

```
var myDaughtersAge = 12;
var myAge = 36;
var ourAgeDifference = myAge - mydaughtersAge;
```

При програмуванні цих інструкцій ви очікуєте, що змінна ourAgeDifference буде містити значення 24 (36 мінус 12). Однак насправді вона буде *невизначеною*. Ім'я змінної не повинно бути mydaughtersAge, воно повинно бути myDaughtersAge .

Замість того, щоб зупинити сценарій та повідомляти про помилку, браузер/інтерпретатор *мовчки* оголошує нову глобальну змінну, яка називається

mydaughtersAge. Оскільки ця змінна невизначена (вона ще не ссилається на значення), будь-які розрахунки, зроблені з цією змінною, також будуть невизначеними. Таким чином, змінна ourAgeDifference тоді також не визначена.

Ці проблеми дуже важко вирішити. На щастя, новий стандарт EMCAScript 5 має так званий *суворий режим*. Коли скрипт інтерпретується в суворому режимі, не можна використовувати змінні, не оголошуючи їх. Якщо ви хочете, щоб скрипт інтерпретувався в строгому режимі, єдине, що вам потрібно зробити, - додати один рядок на початку скрипту:

```
"use strict";
var myDaughtersAge = 12; var myAge = 36;
var ourAgeDifference = myAge - mydaughtersAge;
```

Рядок/інструкція "use strict"; говорить інтерпретатору, що скрипт повинен інтерпретуватися в суворому режимі. Якщо ви зараз спробуєте запустити цей скрипт, веб-браузер зупинить сценарій і повідомить про помилку.

На додаток до перевірки того, чи оголошена змінна перед використанням, строгий режим включає в себе ще пару інших речей, які полегшують написання правильного коду JavaScript. Крім того, цілком імовірно, що нові версії стандарту JavaScript будуть близькі до обмежень синтаксису JavaScript, встановлених строгим режимом.

1.4 Інструкція та Вирази

Якщо ви подивитеся на елементи в синтаксичних діаграмах, ви, напевно, помічаєте, що фрагмент значення або програми в правій частині присвоювання є *виразом*. Так яка різниця між виразом та *інструкцією*? Різниця між ними полягає в тому, що *інструкція* певним чином змінює пам'ять, тоді як *вираз* має значення. Приклади інструкцій – це виклики методу та присвоювання. Інструкції часто використовують вирази. Ось кілька прикладів виразів:

```
16
numberOfBananas
2
a + 4
numberOfBananas + 12 - a
-3
"myCanvas"
```

Всі ці вирази представляють значення певного типу. За винятком останнього рядка, всі вирази - це числа. Останній вираз – це рядок (символів).

1.5 Оператори та складніші вирази

Арифметичні оператори

У виразах, які є числовими, ви можете використовувати наступні арифметичні оператори:

- + додати
- - відняти
- * помножити
- / розділити
- % розділення залишку (вимовляється "Модуль")

Коли використовується оператор ділення /, в деяких випадках результат є дійсним числом (замість цілого числа). Наприклад, після виконання наступної інструкції, змінна у містить значення 0,75:

```
var y = 3/4;
```

Спеціальний оператор % дає залишок ділення. Наприклад, результат 14% 3 становить 2, а результат 456% 10 становить 6. Результат завжди лежить між 0 і значенням праворуч від оператора.

Пріоритет операторів

Коли у виразі використовуються декілька операторів, застосовуються звичайні арифметичні правила пріоритету: множення перед додаванням. результат виразу 1 + 2 * 3, отже, становить 7, а не 9. Додавання та віднімання мають однаковий пріоритет, а також множення та розподілу.

Якщо вираз містить декілька операторів з тим же пріоритетом, то вираз обчислюється зліва направо. Отже, результат 10-5-2 становить 3, а не 7.

Якщо ви хочете відхилитися від цих стандартних правил, ви можете скористатися дужками: наприклад, (1 + 2) * 3 та 3+ (6-5). Використання більшої кількості дужок, ніж потрібно, не заборонено: наприклад, 1+ (2 * 3).

Таким чином, вираз може бути постійним значенням (наприклад, 12), це може бути змінною, може бути іншим виразом у дужках, або це може бути вираз, за яким слідує оператор, а потім інший вираз. На малюнку 1.5 показана (часткова) синтаксична діаграма, що представляє вираз.

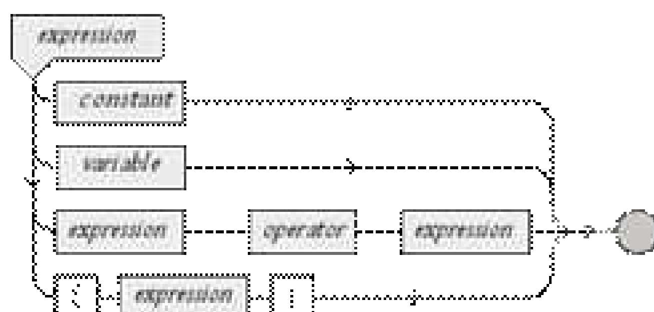


Рис. 1.5 *Часткова синтаксична схема виразу*

Призначення функції для змінної

У JavaScript функції (групи інструкцій) зберігаються в пам'яті. Функції самі є виразами. Отже, можна призначити функцію змінній. Наприклад:

```
var someFunction = function () {
// робити щось
Курс лекцій
«Створення ігрових веб-додатків»
9 | 73
}
```

У цьому прикладі оголошується змінна someFunction і призначається їй значення. Значення, на яке ссилається ця змінна, є *анонімною функцією*. Якщо ви хочете виконати інструкції, що містяться в цій функції, ви можете викликати це, використовуати назву змінної, таким чином:

```
someFunction();
```

Яка різниця між таким способом визначення функції та способом, який ви вже бачили?

```
function someFunction () {
// робити щось
}
```

Насправді, різниця не велика. Головне, що, визначаючи функцію традиційним способом (не використовуючи змінну), функція не повинна бути визначена, перш ніж вона може бути використана. Коли браузер інтерпретує файл JavaScript, він робить це в два етапи. На першому етапі браузер створює список доступних функцій. На другому етапі браузер інтерпретує залишок скрипту. Це необхідно тому, що для правильної інтерпретації сценарію браузеру потрібно знати, які функції доступні.

Наприклад, цей фрагмент коду JavaScript буде працювати відмінно, навіть якщо функція визначена після її виклику:

```
someFunction();
function someFunction () {
// do something
}
```

Однак, цей фрагмент коду призведе до помилки:

```
someFunction();
var someFunction = function () {
// do something
}
```

Браузер скаржиться на те, що сценарій отримує доступ до змінної someFunction, яка ще не була оголошена. Виклик функції після його визначення є ідеальним:

```
var someFunction = function () {
// do something
}
someFunction();
```

Змінні, які складаються з декількох значень

Замість того, щоб містити одне значення, змінна може також складатися з *кількох значень.* Це схоже на те, що ви виконуєте в функції, яка об'єднує команди. Наприклад:

```
function mainLoop () { canvasContext.fillStyle = "blue";
canvasContext.fillRect(0, 0, canvas.width, canvas.height); update();
draw();
window.setTimeout(mainLoop, 1000 / 60);
}
```

Ви можете виконати всі ці інструкції, викликаючи функцію mainLoop. Інструкції, що належать до функції, згруповані за допомогою фігурних дужок. Подібно інструкціям групування, ви також можете групувати змінні в більшу змінну. Ця більша змінна тоді містить декілька значень. Подивіться на наступний приклад:

```
var gameCharacter = { name : "Merlin", skill : "Magician", health : 100,
power : 230
};
```

Це приклад *складної змінної*. Змінна gameCharacter складається з кількох значень, кожне з яких має ім'я та значення, до якого посилається ім'я. Отже, в певному сенсі змінна gameCharacter *складається з інших змінних.*

Малюнок 1.6 демонструє (часткову) синтаксичну діаграму діаграма об'єкта літерального виразу

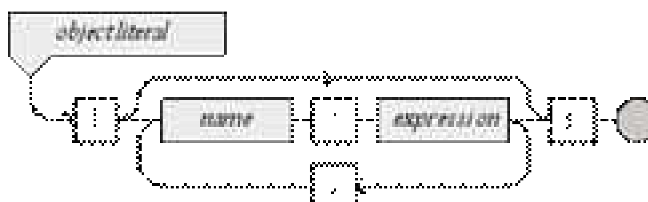


Рис. 1.6 *(Часткова) синтаксична діаграма об'єкта літерального виразу*

Після оголошення та ініціалізації змінної gameCharacter , пам'ять буде виглядати так, як це зображено на малюнку 1.7 .

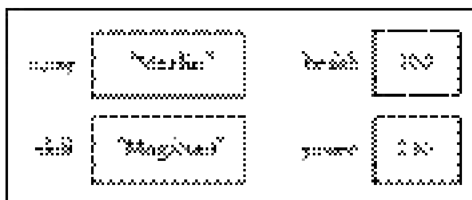


Рис.1.7 *Структура пам'яті після створення складної змінної*

Ви можете отримати доступ до даних у складній змінній наступним чином:

```
gameCharacter.name = "Arjan";
var damage = gameCharacter.power * 10;
```

Як ви бачите, ви можете отримати доступ до змінних, які є частиною gameCharacter, написавши своє ім'я після крапки. JavaScript навіть дозволяє змінювати структуру складної змінної після того, як ви її оголосили та ініціалізували. Наприклад, подивіться на наступний код:

```
var anotherGameCharacter = { name : "Arthur",
skill : "King", health : 25,
power : 35000
};
anotherGameCharacter.familyName = "Pendragon";
```

Змінна anotherGameCharacter тепер складається з п'яти частин: name, skill, health, power, and familyName.

Оскільки змінні можуть також вказувати на функції, ви навіть можете включити підменю, яке вказує на функцію. Наприклад, ви можете визначити anotherGameCharacter наступним чином:

```
var anotherGameCharacter = { name : "Arthur", familyName :
"Pendragon", skill : "King",
health : 25,
power : 35000,
healMe : function () { anotherGameCharacter.health = 100;
}
};
```

Як і раніше, ви можете додати частину функції до змінної після того, як їй було призначено значення:

```
anotherGameCharacter.killMe = function ()
{ anotherGameCharacter.health = 0;
```

};

Ви можете викликати ці функції так само, як і інші змінні. Наступна інструкція повністю відновлює здоров'я персонажа гри:

```
anotherGameCharacter.healMe();
```

І якщо ви хочете вбити персонажа, то інструкція GameCharacter.killMe (); інструкція зробить цю роботу. Ви можете групувати пов'язані дані та функції разом. У цьому прикладі групують змінні, які всі належать до того самого ігрового символу. Він також додає кілька функцій, корисних для цього персонажа гри. Відтепер, якщо функція належить до змінної, будемо називати цю функцію *методом*. Змінні, що складаються з інших змінних, будемо називати *об'єктами*. І якщо а змінна є частиною об'єкту, будемо називати цю змінну *змінною членом*.

Ви, напевно, можете уявити, наскільки потужні об'єкти та методи. Вони забезпечують спосіб приведення структури в складний ігровий світ.

Шляхом групування змінних в об'єктах і надаючи методи, що належать до цих об'єктів, ви можете писати програми, які набагато легше зрозуміти.

1.6 Гра MovingSquare

Розглянемо просту програму, яка переміщає квадрат над полотном. Її мета - проілюструвати дві речі:

- Як частини update, draw ігрового циклу працюють більше детально
- Як використовувати об'єкти

Перш ніж почати писати цю програму, давайте розглянемо код основного прикладу:

```
var canvas = undefined;
var canvasContext = undefined;
function start () {
canvas = document.getElementById("myCanvas");
canvasContext = canvas.getContext("2d");
mainLoop();
}
document.addEventListener('DOMContentLoaded', start);
function update () {
}
function draw () {
canvasContext.fillStyle = "blue";
canvasContext.fillRect(0, 0, canvas.width, canvas.height);
}
function mainLoop () {
update();
```

```
draw();
window.setTimeout(mainLoop, 1000 / 60);
}
```

Тут є кілька декларацій змінних та декілька функцій, які щось роблять з цими змінними. З новими знаннями про групування змінних разом у об'єктах давайте вияснимо, що всі ці змінні та функції належать до *ігрової* програми, а саме:

```
"use strict"; var Game = {
canvas : undefined,
canvasContext : undefined
};
Game.start = function () {
Game.canvas = document.getElementById("myCanvas");
Game.canvasContext = Game.canvas.getContext("2d");
Game.mainLoop();
};
document.addEventListener('DOMContentLoaded', Game.start);
Game.update = function () {
};
Game.draw = function () {
Game.canvasContext.fillStyle = "blue";
Game.canvasContext.fillRect(0, 0, Game.canvas.width,
Game.canvas.height);
};
Game.mainLoop = function () { Game.update(); Game.draw();
window.setTimeout(mainLoop, 1000 / 60);
};
```

Головне, що ви робите тут, - створюєте єдину складну змінну (об'єкт), яку називають Game. Цей об'єкт має два *змінні-члени*: canvas і canvasContext.

Крім того, ви додаєте до цього об'єкта кілька *методів* , включаючи методи, які разом утворюють ігровий цикл. Ви визначаєте методи, що належать цьому об'єкту окремо (іншими словами, вони не є частиною декларації змінної та початкового призначення). Причиною є те, що тепер ви можете легко розрізнити *дані,* об'єкт яких складається з *методів, які щось роблять з даними.* Зауважте також, що ви додаєте інструкцію "use strict"; до програми.

Ви використовуєте змінну rectanglePosition для зберігання бажаної x-позиції прямокутника.У методі draw ви можете використовувати це значення, щоб намалювати прямокутник десь на екрані.У цьому прикладі ви намалюєте менший прямокутник, який не покриває весь полотно, так що ви можете побачити його переміщення.

Це новий метод малювання:

```
Game.draw = function () {
Game.canvasContext.fillStyle = "blue";
Game.canvasContext.fillRect(Game.rectanglePosition, 100, 50, 50);
}
```

Тепер єдине, що вам потрібно зробити, - обчислити, яким має бути x-позиція прямокутника. Ви це робите в методі оновлення, оскільки зміна x-позиції прямокутника означає, що ви *оновлюєте ігровий світ*. У цьому простому прикладі давайте змінимо положення прямокутника на основі пройденого часу. У JavaScript ви можете скористатися двома наступними інструкціями, щоб отримати поточний системний час:

```
var d = new Date();
var currentSystemTime = d.getTime();
```

Ви не бачили типи позначень, використаних у першому рядку раніше. До цих пір припустимо, що new Date() створює складну змінну (об'єкт), яка заповнюється інформацією про дату та час, а також парою корисних методів. Одним з таких методів є getTime. Ви викликаєте цей метод для об'єкта d і зберігаєте його результат у змінній currentSystemTime. Ця змінна тепер містить кількість мілісекунд, що минули з 1 січня 1970 року (!). Ви можете уявити, що це число досить велике. Ви розділите системний час на ширину полотна, візьміть залишок цього розділу та використовуйте його як x-позицію прямокутника. Таким чином, ви завжди отримуєте x-позицію між нулем і шириною полотна. Ось повний метод оновлення, який робить це:

```
Game.update = function () { var d = new Date();
Game.rectanglePosition = d.getTime() % Game.canvas.width;
};
```

Як ви знаєте, методи update, draw викликаються послідовно, приблизно 60 разів на секунду. Кожного разу, коли це відбувається, системний час змінився (оскільки минув час), що означає, що положення прямокутника буде змінено, і воно буде намальовано в іншому місці, ніж раніше.

Вам потрібно зробити ще одну річ, перш ніж цей приклад буде працювати як належить. Якщо б ви запустили програму, подібну до цього, на екрані з'явиться синя панель. Причина в тому, що ви наразі малюєте новий прямокутник на вершині старого. Для того, щоб вирішити це, вам потрібно *очистити полотно* кожного разу, перш ніж намалювати його знову.

Очищення полотна здійснюється методом clearRect. Цей метод очищає прямокутник заданого розміру і все, що було намальовано в ньому. Наприклад, це інструкція очищає полотно:

```
Game.canvasContext.clearRect(0, 0, Game.canvas.width,
Game.canvas.height);
```

Для зручності ви розміщуєте цю інструкцію в методі clearCanvas, як зазначено нижче:

```
Game.clearCanvas = function () {
Game.canvasContext.clearRect(0, 0, Game.canvas.width,
Game.canvas.height);
};
```

Єдине, що вам потрібно зробити - переконайтеся, що визвано update та draw. Ви робите це в методі mainLoop :

```
Game.mainLoop = function () {
Game.clearCanvas ();
Game.update ();
Game.draw ();
window.setTimeout (Game.mainLoop, 1000/60);
```

Тепер приклад завершений! Ви можете запустити цю програму, двічі натиснувши на MovingSquare.html файл у папці, що належить до цього розділу.

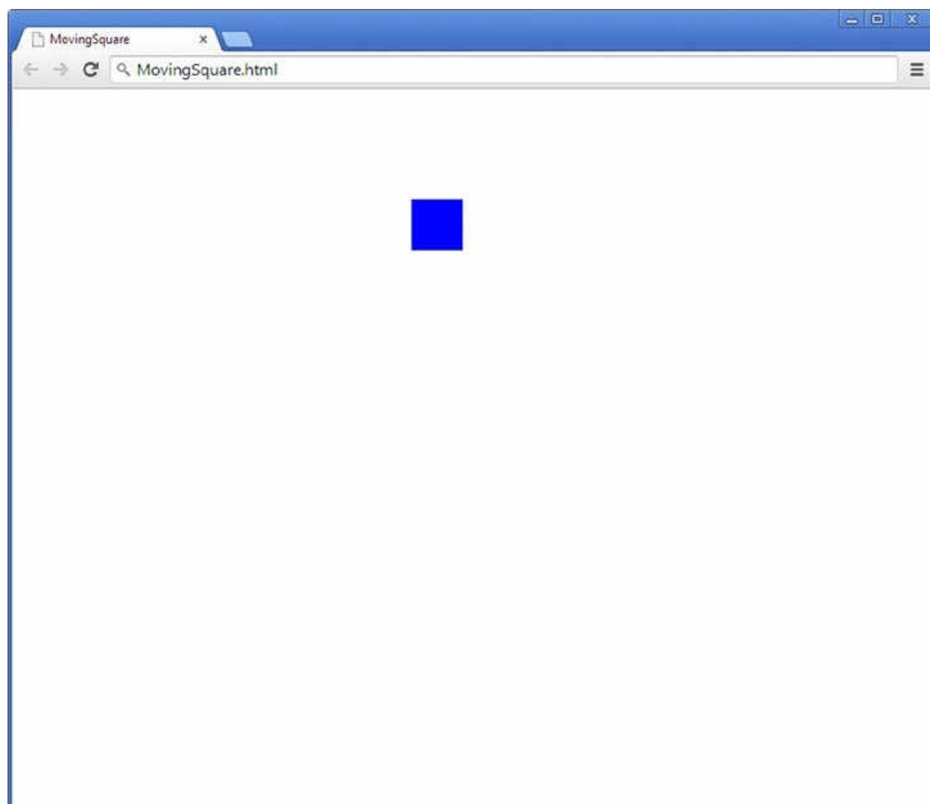


Рис. 1.8 *Вихід із прикладу MovingSquare*

Подивіться, як змінюється положення прямокутника залежно від часу.

Область видимості змінних

Місце, де ви оголошуєте змінну, впливає на те, де ви можете використовувати цю змінну. Подивіться на змінну d в програмі MovingSquare. Ця змінна оголошується (і призначається значенням) у методі update. Оскільки це оголошено в методі update , ви можете використовувати його лише в цьому методі. Наприклад, вам не дозволяється використовувати цю змінну знову в методі draw. Звичайно, ви можете оголосити іншу змінну d в методі draw, але важливо розуміти, що змінна d, оголошена в update, в цьому випадку не може бути такою ж d змінною, оголошеною в методі в draw.

Крім того, якщо ви оголошувати змінну на рівні *об'єкта*, ви можете використовувати її в будь-якому місці, доки ви покладете назву об'єкта перед ним. Місця, де можна використовувати змінну разом називається *областю видимості* змінної. У цьому прикладі область видимості змінної d є метод update, а область видимості змінної Game.rectanglePosition є глобальною.

Питання для перевірки знань.

1. Які теги HTML вам відомі?
2. Які елементи HTML5 вам відомі?
3. Як створити полотно в HTML?
4. Як з JavaScript звернутися до полотна і змінити його розміри?
5. Як відобразити криву (прямокутник) на полотні?
6. Як вивести на полотно зображення з файлу?
7. Яка вбудована функція JavaScript створює інтервал запуска, необхідний для анімації?
8. Для чого необхідні методи save і restore контексту полотна?

Завдання.

1. Реалізувати переміщення прямокутника на полотні, швидкість збільшується та зменшується в арифметичній прогресії.
2. Намалюйте на полотні будинок з трикутним дахом, вікном і дверима, розфарбуйте його різними кольорами.
3. Відкрийте малюнок на полотні в трьох варіантах: масштаб 1:1, зменшений в 2 рази, збільшений в 2 рази.
3. Доопрацювати анімацію з приклада в лекції, щоб прямокутник

рухався не тільки зліва направо, потім зникав і знову рухався зліва направо, а

й виконував послідовні нескінченні рухи: зліва направо, потім справа наліво,

потім знову зліва направо і т. д.

4. Напишіть сценарій переміщення кольорового квадрата по колу.

Траєкторію зручно описувати параметричними рівняннями:

x = R * cos (t),

y = R * sin (t),

де: R- радіус кола, 0 <= t <= 2

Лабораторна робота 2
Робота з покажчиком миші мовою JavaScript на прикладі гри Painter

Анотація. Лабораторна робота орієнтована на оволодіння студентами навичок роботи зі спрайтами мовою програмування JavaScript.

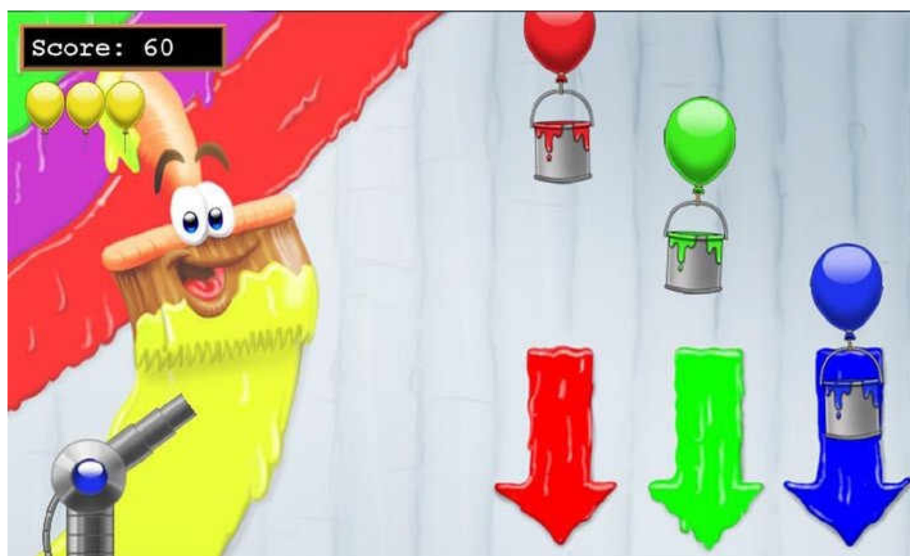
Мета лабораторної роботи:

- Оволодіти прийомами додавання декількох спрайтів в ігровий веб-додаток.
- Освоїти навички сворення ігрового циклу для переміщення спрайтів.
- Освоїти навички створення методу update для розрахунку позиції для відображення спрайту, та методу draw для відображення.
- Навчитися додавати звукові ефекти в ігрові веб-додатки.

У разі успішного виконання лабораторної роботи студент буде вміти додавати динамічні спрайти в ігровий веб-додаток.

2.1 Створення гри

У цій лабораторній роботі ви починаєте розробляти гру під назвою *Painter* (див. малюнок 2.1). Під час розробки цієї гри ви познайомитесь з кількома новими прийомами, які дуже корисні при програмуванні ігор, такі як організація інструкцій у класах та методах, умовних інструкціях, ітерації та багато іншого.



© Springer-Verlag Berlin Heidelberg 2013

Рисунок 2.1 *Гра Painter*

Метою гри Painter є збирання трьох різних кольорів фарби: червоного, зеленого та синього. Фарба падає із неба в банки, які тримаються плавучими повітряними кульками, і ви повинні переконатися, що кожна банка може

володіти потрібним кольором, перш ніж він падає в нижній частині екрана.

Ви можете змінити колір фарби шляхом зйомки кульки фарби потрібного кольору на падаючому балоні. Ви можете вибрати той колір, яким ви стріляєте за допомогою клавіш R, G і B на клавіатурі. Ви можете стріляти по м'ячу, клацнувши лівою кнопкою миші на екрані гри.Натиснувши ще далі від гармати, ви дасте м'ячу вище швидкість Місце, де ви натискаєте, також визначає напрямок, в якому вистрілює гармата. За кожний правильний контейнер, ви отримуєте 10 балів. За кожну неправильно забарвлену банку ви втрачаєте життя. У цій грі Painter вам потрібно показати спрайти, які рухаються на екрані.

2.2 Отримання позиції миші

Подивіться на програму Balloon1 у конспекті лекцій. Існує невелика різниця між нею і програмою FlyingSprite, де ви обчислите позицію кульки за допомогою системного часу:

```
var d = new Date();
Game.balloonPosition.x  = d.getTime() * 0.3 % Game.canvas.width;
```

Позиція, яку ви обчислюєте, зберігається в змінній balloonPosition. Тепер ви хочете створити програму, де замість того, щоб обчислюватися на основі пройденого часу, позиція кульки така ж, як і поточна позиція миші.

Отримання поточної позиції миші дуже просто завдяки *події*.

У JavaScript ви можете обробляти різні типи подій. Приклади подій такі:

- Гравець рухає мишею
- Гравець натискає ліву клавішу
- Гравець натискує кнопку
- Сторінка HTML була завантажена
- Повідомлення надійшло з мережі з'єднання
- Спрайт закінчив завантаження

Коли виникає така подія, ви можете вибрати виконання інструкцій. Наприклад, коли гравець рухає мишею, ви можете виконати декілька інструкцій, які отримують нове розташування миші, і зберігати його в змінній, щоб ви могли використовувати його для відмальовки спрайта в цій позиції. Кілька JavaScript-об'єктів допоможуть вам зробити це. Наприклад, коли ви виводите HTML-сторінку, змінна документа дає вам доступ до всіх елементів на сторінці. Але, що більш важливо, ця змінна також дозволяє отримати доступ до способу взаємодії користувача з документом за допомогою миші, клавіатури або торкання екрану.

Ви вже використовували цю змінну кількома способами. Наприклад, тут ви використовуєте документ для отримання елемента canvas з HTML-сторінки:

```
Game.canvas = document.getElementById ("myCanvas");
```

Крім getElementById, в об'єкті document є багато інших методів та змінних-членів. Наприклад, є змінна-член, викликана onmousemove, якій ви можете призначити значення. Ця змінна-член не посилається на числове значення або рядок, а посилається на до функцію/метод. Кожного разу, коли миша переміщується, браузер викликає цю функцію. Потім ви можете написати інструкції до функції, яка обробляє подію будь-яким способом.Через це ці функції називаються *обробники подій*. Використання функцій обробників подій є дуже ефективним способом обробки вводу. Інший підхід полягає в тому, щоб поставити інструкції в циклі гри, які отримують поточну позицію миші або ключі, які зараз натискаються в кожній ітерації. Хоча це буде працювати, це буде набагато повільніше, ніж використання обробників подій, тому що вам доведеться перевіряти на введення на кожній ітерації, а не тільки тоді, коли гравець дійсно щось робить.

Функція обробника події має певний заголовок. Він містить один параметр, який, коли функція викликається, містить об'єкт, що надає інформацію про події. Наприклад, тут є порожня функція обробки подій:

```
function handleMouseMove (evt) {
// do something here
}
```

Як ви бачите, у функції є один параметр evt, який буде містити інформацію про подію, що потребує обробки. Тепер ви можете призначити цю функцію змінній onmousemove:

```
document.onmousemove = handleMouseMove;
```

Тепер, кожного разу, коли миша переміщується, функція handleMouseMove викликається. Ви можете поставити інструкції в цій функції, щоб витягти позицію миші з об'єкта evt. Наприклад, ця функція обробки подій витягує x-позицію миші та y-позицію і зберігає їх у змінній balloonPosition :

```
function handleMouseMove(evt) {
Game.balloonPosition = { x : evt.pageX, y : evt.pageY };
}
```

Змінні-члени pageX і pageY об'єкта EVT містять позицію миші відносно сторінки, тобто верхній лівий кут сторінки має координати (0, 0).

Ви можете побачити кілька прикладів розташування миші на малюнку 2.1: три кути позначені їх відповідними позиціями, як це було повідомлено під час запуску програми в браузері.

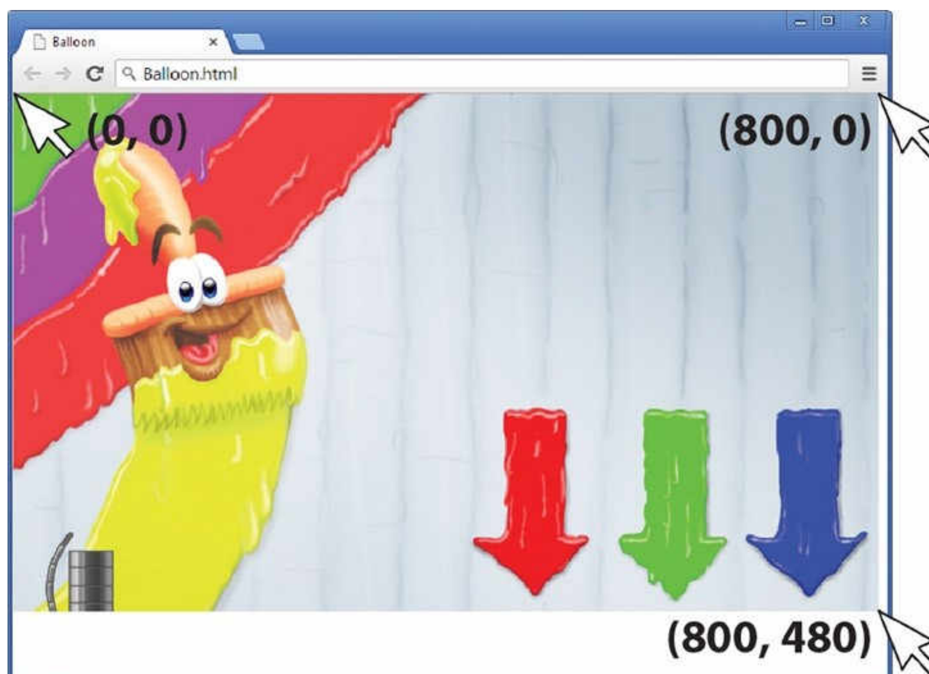


Рис. 2.1 *Позиції миші для верхнього лівого, верхнього та нижнього правого кутів*

Оскільки метод Draw просто намалює кулю в позиції миші, куля тепер йде за мишкою. На малюнку 2.2 показано, що це схоже. Ви можете побачити кулю, намальовану під вказівником миші; він відстежуватиме покажчик, коли ви переміщуєте його.

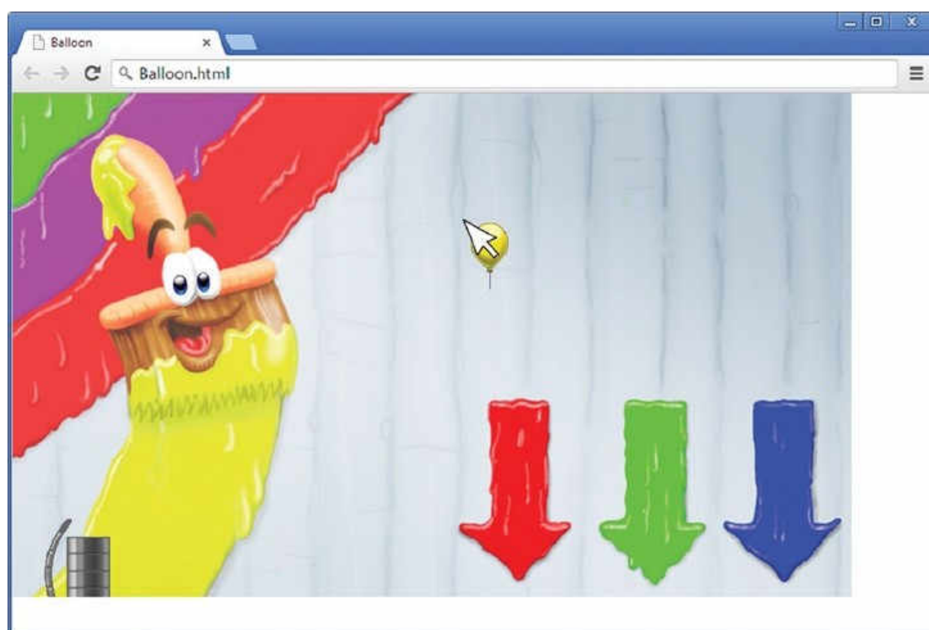


Рис.2.2 *Знімок екрана програми Balloon1*

На рисунку 2.2 ви можете побачити, що куля не розташовується в центрі під самим кінцем покажчика. Це є причиною, і в наступному розділі детально розглядається ця проблема. На даний момент, просто пам'ятайте, що спрайт

розглядається як прямокутник. Верхній лівий кут вирівнюється з кінчиком вказівника. Куля з'являється неправильно, тому що куля кругла і не поширюється на кути прямокутника.

Замість сторінки X та сторінки Y ви також можете використовувати clientX та clientY, що також дає позицію миші. Проте, clientX та clientY не беруть до уваги прокручування. Припустимо, що ви обчислили позицію миші таким чином:

```
Game.balloonPosition = { x : evt.clientX, y : evt.clientY };
```

На малюнку 2.3 показано, що зараз може не спрацювати. Через прокручування значення clientY менше, ніж 480, навіть якщо миша розташована внизу фонового зображення.В результаті повітряна куля більше не намальована в позиції миші. Тому пропонується використовувати сторінку X і pageY, а не clientX та clientY . У деяких випадках може бути корисним не приймати прокрутку до уваги, наприклад, якщо ви розробляєте оголошення, яке відображається посередині вікна браузера, навіть якщо користувач намагається прокрутити його мимо.

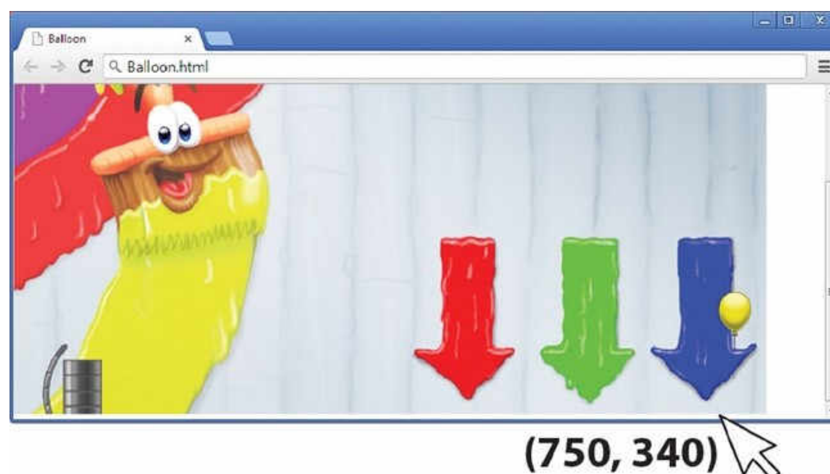


Рис. 2.3 *y-позиція є 340 (замість від 480) тому що **clientY** не приймає прокрутку до уваги*

2.3 Зміна джерела спрайта

Коли ви запускаєте приклад Balloon1, зверніть увагу, що куля намальована так, що верхній лівий кут справа знаходиться у поточній позиції миші. Коли ви малюєте спрайт у певній позиції, поведінка за умовчанням полягає в тому, що верхній лівий кут справа втягується в цю позицію. Якщо ви виконуєте наступну інструкцію Game.drawImage (someSprite, somePosition); спрайт під назвою somesprite малюється на екрані так, що його верхній лівий кут знаходиться в позиції somePosition. Ви також можете назвати верхній лівий кут спрайту його *джерелом*. Джерело можна змінювати. Наприклад, припустимо, ви хочете намалювати центр справа someSprite на позицію somePosition. Ви можете

обчислити його, використовуючи змінні ширини та висоти типу зображення. Оголосимо змінну з назвою origin і збережемо в ній центр спрайта:

```
var origin = {x: someSprite.width / 2, y: someSprite.height / 2};
```

Тепер, якщо ви хочете намалювати sprite someSprite з іншим джерелом, ви можете зробити це наступним чином:

```
var pos = {x: somePosition.x - origin.x,
y: somePosition.y - origin.y};
Game.drawImage (someSprite, pos);
```

Положення somePosition вказує центр спрайту. Замість того, щоб самостійно обчислити позицію по відношенню до самого початку, метод drawImage з контексту полотна також має можливість вказати зміщення джерела. Ось приклад:

```
Game.canvasContext.save ();
Game.canvasContext.translate (position.x, position.y);
Game.canvasContext.drawImage (sprite, 0, 0, sprite.width, sprite.height,
-origin.x, -origin.y, sprite.width, sprite.height);
Game.canvasContext.restore ();
```

У цьому прикладі першим кроком є збереження поточного статусу малюнка. Потім ви застосовуєте перетворення. Потім ви називаєте drawImage метод, в якому вам необхідно надати кілька різних параметрів: який спрайт буде намальований і (з використанням чотирьох параметри) яку частину спрайту слід намалювати. Ви можете зробити це, вказавши верхню ліву координату спрайта та розмір частини прямокутника, яку слід намалювати. У цьому простому випадку ви хочете намалювати весь спрайт, тому верхня ліва координата - точка (0, 0). Ви малюєте прямокутну частину, яка має ту саму ширину та висоту, що і весь спрайт. Це також показує, що можна використовувати цю функцію для зберігання декількох спрайтів в одному файлі зображення, коли треба завантажувати цей файл у пам'ять лише один раз.

В залежності від джерела, яке ви вибрали, результат відрізняється.На малюнку 2.4 наведено два приклади

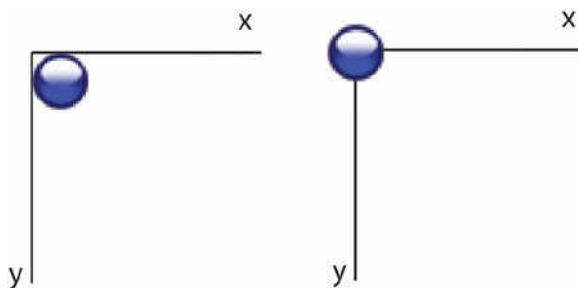


Рис. 2 4 Малювання спрайту в позиції (0, 0) з джерелом в верхньому лівому куті (ліворуч) та в в центрі спрайта (праворуч)

Як ви, напевно, помітили, вилучення однієї позиції з іншої є дещо громіздкою дією в JavaScript:

```
var pos = {x: somePosition.x - origin.x,
y: somePosition.y - origin.y};
```

Було б набагато приємніше, якщо б ви могли написати щось на зразок цього:

```
var pos = somePosition - origin;
```

На жаль, це неможливо в JavaScript. Деякі мови програмування (наприклад, Java та C #) підтримують перевантаження оператора. Це дозволяє програмісту визначити, що має відбутися, наприклад, коли два об'єкти «додаються» один до одного за допомогою оператора плюс .Проте все не втрачено.Можна визначити методи , які виконують ці арифметичні операції над литералов об'єктів , таких як той , визначеної вище.

Тепер, коли ви знаєте, як намалювати спрайт на іншому джерелі, ви можете, наприклад, намалювати кульку таким чином, щоб його нижній центр був прикріплений до покажчика миші. Щоб побачити це в дії, подивіться на програму Balloon2. Ви декларуєте додаткову змінну-член, в якій ви зберігаєте джерело кулі:

```
Var Game = {
canvas: undefined,
canvasContext: undefined,
backgroundSprite: undefined,
balloonSprite: undefined
mousePosition: {x: 0, y: 0},
balloonOrigin: {x: 0, y: 0}
};
```

Ви можете тільки обчислити джерело, коли спрайт завантажений:

```
Game.balloonOrigin = {x: Game.balloonSprite.width / 2,
y: Game.balloonSprite.heig ht};
```

Джерело встановлюється на половину ширини спрайту, але до повної висоти. Іншими словами, це джерело є нижнім центром спрайту. Розрахунок джерела в методі draw не є ідеальним; було б краще, якщо б ви могли обчислити джерело лише один раз, одразу після завантаження зображення.

Пізніше ви побачите кращий спосіб зробити це. Тепер ви можете поширіати метод drawImage в об'єкті Game, щоб він підтримував выдображення спрайта в іншому джерелі. Єдине, що вам потрібно зробити для цього, - додати додатковий параметр позиції і передати значення x і y цього параметра в метод drawImage контексту полотна:

```
Game.drawImage = function (sprite, position, origin) {
Game.canvasContext.save();
Game.canvasContext.translate(position.x, position.y);
Game.canvasContext.drawImage(sprite, 0, 0, sprite.width, sprite.height,
-origin.x, -origin.y, sprite.width, sprite.height);
Game.canvasContext.restore();
};
```

У методі draw ви можете тепер обчислити джерело і передавати його методу drawImage , як показано нижче:

```
Game.draw = function () {
Game.drawImage(Game.backgroundSprite, { x : 0, y : 0 }, { x : 0, y : 0 });
Game.balloonOrigin = { x : Game.balloonSprite.width / 2,
y : Game.balloonSprite.height };
Game.drawImage(Game.balloonSprite,Game.mousePosition,
Game.balloonOrigin);
};
```

2.4 Використання позиції миші для повороту гармати

Однією з особливостей гри Painter є те, що вона містить гармату, яка обертається відповідно до положення миші. Ця гармата контролюється гравцем для того, щоб стріляти в кольорові кулі. Ви можете написати частину програми, яка це робить, використовуючи інструменти, описані в цій роботі. Ви можете бачити цю роботу в прикладі Painter1.

Щоб зробити це можливим, вам слід декларувати кілька змінних членів. По-перше, вам потрібні змінні для зберігання фону та спрайту гармати. Вам також потрібно зберегти поточну позицію миші, як і в попередніх прикладах цього розділу. Ви повинні зберегти свою позицію, її джерело та поточне обертання. Нарешті, вам потрібно полотно та контекст полотна, щоб ви могли малювати об'єкти гри. Як завжди, всі ці змінні оголошуються сленами об'єкта game:

```
var Game = {
canvas : undefined,
canvasContext : undefined,
backgroundSprite : undefined,
```

```
cannonBarrelSprite : undefined,
mousePosition : { x : 0, y : 0 },
cannonPosition : { x : 72, y : 405 },
cannonOrigin : { x : 34, y : 34 },
cannonRotation : 0
};
```

Як позиції, так і джерелу стволу гармати призначаються значення, коли визначається змінна Game. Позиція ствола вибирається таким чином, щоб вона добре підходила до гармати, яка вже була намальована. Зображення ствола містить круглу частину та ствол. Ствол повинен оберталася навколо центру круглої частини. Це означає, що ви повинні встановити цей центр як джерело. Оскільки частина кола знаходиться ліворуч на спрайті, а радіус цього кола становить половину висоти гарматного ствола (що дорівнює 68 пікселів), ви встановлюєте початкове положення ствола 34, 34. Щоб намалювати ствол гармати під кутом, ви повинні застосувати обертання при малюванні спрайту гарматного ствола на екрані.Це означає , що ви повинні поширити метод DrawImage таким чином, щоб він враховував обертання. Обертання враховується з допомогою методу rotate, який є частиною контексту полотна.

Ви також можете додати параметр методу DrawImage , який дозволяє визначити кут , при якому об'єкт повинен бути повернений. Нова версія методу DrawImage виглядає наступним чином :

```
Game.drawImage = function (sprite, position, rotation, origin) {
Game.canvasContext.save();
Game.canvasContext.translate(position.x, position.y);
Game.canvasContext.rotate(rotation);
Game.canvasContext.drawImage(sprite, 0, 0, sprite.width, sprite.height,
-origin.x, -origin.y, sprite.width, sprite.height);
Game.canvasContext.restore();
};
```

У методі start ви завантажуєте два спрайти:

```
Game.backgroundSprite = new Image();
Game.backgroundSprite.src = "spr_background.jpg";
Game.cannonBarrelSprite = new Image();
Game.cannonBarrelSprite.src = "spr_cannon_barrel.png";
```

Наступним кроком є реалізація методів у циклі гри. До цього часу метод update завжди був порожнім. Тепер у вас є вагомі підстави використовувати його. У методі update ви оновлюєте ігровий світ, який у цьому випадку означає, що ви обчислите кут, за яким буде намальована гармата .

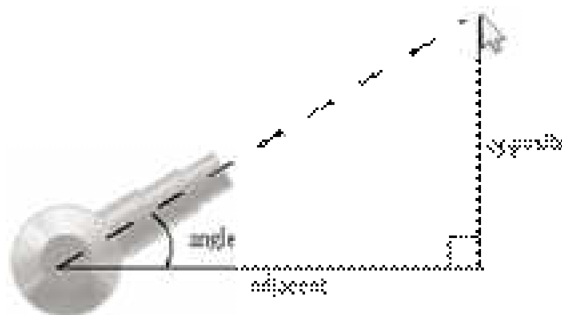


Рис. 2.5 *Розрахунок кута ствола на основі позиції покажчика миші*

Ви можете обчислити довжину протилежної та сусідньої сторони, обчисливши різницю між поточним положенням миші та позицією гармати, як зазначено нижче:

```
var opposite = Game.mousePosition.y - Game.cannonPosition.y;
var adjacent = Game.mousePosition.x - Game.cannonPosition.x;
```

JavaScript має об'єкт Math , який може допомогти. Об'єкт Math містить ряд корисних математичних функцій, в тому числі тригонометричних функцій, такі як синус, косинус, арксинус, арккосинус і арктангенс. Дві функції в об'єкті Math обчислюють арктангенс. Перша версія приймає одне значення як параметр. Ви не можете використовувати цю версію в цьому випадку: коли миша знаходиться безпосередньо над стволом, відбувається ділення на нуль:

```
Game.cannonRotation = Math.atan2(opposite, adjacent);
```

Ці інструкції поміщені в update .Ось повний метод:

```
Game.update = function () {
var opposite = Game.mousePosition.y - Game.cannonPosition.y;
var adjacent = Game.mousePosition.x - Game.cannonPosition.x;
Game.cannonRotation = Math.atan2(opposite, adjacent);
};
```

Єдине , що залишилося зробити, це намалювати спрайт на екрані в методі draw, в правильному положенні і під правильним кутом:

```
Game.draw = function () { Game.clearCanvas();
Game.drawImage(Game.backgroundSprite, { x : 0, y : 0 }, 0,
{ x : 0, y : 0 });
Game.drawImage(Game.cannonBarrelSprite, Game.cannonPosition,
Game.cannonRotation, Game.cannonOrigin);
};
```

Завдання.
Модифікуйте гру Painter, перемістивши спрайт з гарматою в правий нижній кут полотна.

Питання для перевірки знань.
1. Що таке подія?
2. Що таке обробник подій?
3. Наведить приклади подій, які можуть бути оброблені мовою JavaScript.
4. Як отримати позицію миші з використанням обробника подій?
5. Що таке джерело спрайту?
6. Яким чином можна використати подію руху миші?
7. Як намалювати спрайт під кутом?
8. Як змінити кут спрайта на основі позиції миші?

Лабораторна робота 3
Створення ігрового веб-додатка з додаванням динамічніх об’єктів

Анотація. Лабораторна робота орієнтована на оволодіння студентами прийомами створення ефективного ігрового цикла мовою JavaScript, створення фіксованого та змінного часового крока, додавання динамічних об’єктів.

Мета лабораторної роботи:
 Освоїти прийоми створення ефективного ігрового цикла.
 Навчитися створювати фіксований та змінний часовий крок.
 Створити ігровий веб-додаток з додаванням динамічного об’єкта.
У разі успішного виконання лабораторної роботи студент буде вміти створювати ігровий веб-додаток з динамічними об’єктами.

3.1 Додавання м'яча до гри

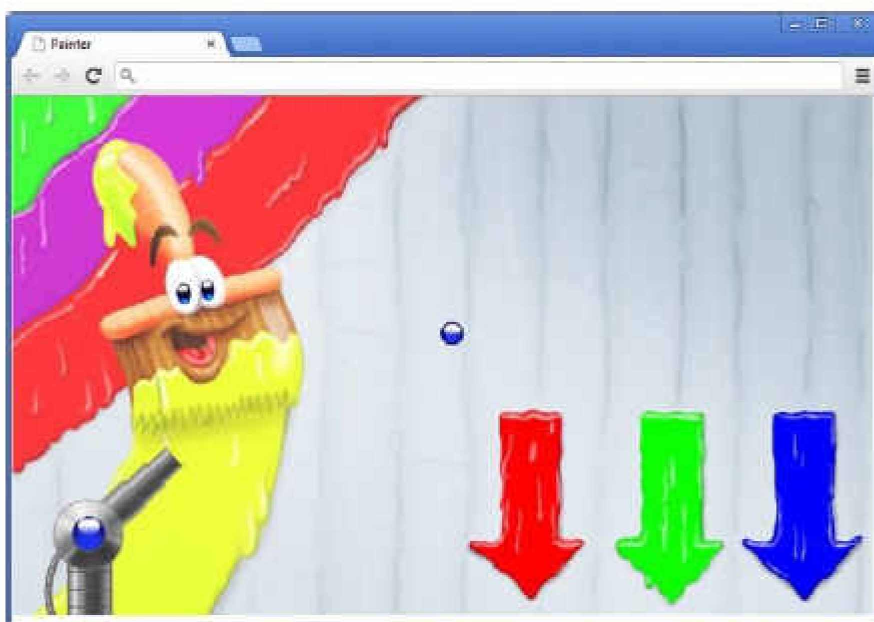
В цій лабораторній роботі ви продовжуєте гру Painter, додавши кулю, якою стріляє об'єкт cannon. Для цього ви додаєте об'єкт ball.

Ви налаштовуєте об'єкт ball у спосіб, дуже схожий на об'єкт cannon. У прикладі Painter4 знаходиться версія гри Painter, яка додає м'яч до ігрового світу (див. малюнок 3.1). Кулька може бути розстріляна з гармати, шляхом натискання на будь-якому місці на екрані гри. Крім того, м'яч змінює колір разом з гарматою. Ви може знайти код, який описує об'єкт ball, в файлі Ball.js. Так само, як об'єкт cannon, ball складається з ряду змінних, таких як позиція, поточний колір м’яча та початкове місце знаходження спрайту.

Оскільки м'яч рухається, вам також потрібно зберігати його *швидкість*. Ця швидкість є вектором, який визначає, як змінюється положення м'яча з часом. Наприклад, якщо м'яч має швидкість (0,1), то кожну секунду його y-позиція збільшується на 1 піксель (тобто м'яч падає). Нарешті, м'яч може бути в двох

станах: він або летить в повітрі, якщо ним вистрілили з гармати, або він в стані очікування (не рухається). Для цього ви додаєте до об'єкта ball додаткову логічну змінну shooting. Це повне визначення структури об'єкту ball:

```
var ball = {
};
ball.initialize  = function() {
ball.position = { x : 65, y : 390 };
ball.velocity = { x : 0, y : 0 };
ball.origin = { x : 0, y : 0 };
ball.currentColor = sprites.ball_red; ball.shooting = false;
};
```



3.1. Скріншот *екрана з прикладом Painter4, який містить гармату та літаючий м'яч*

У іграх, розроблених вами в даних лабораторних роботах, більшість об'єктів мають позицію та швидкість. Оскільки роботи стосуються лише 2D-ігор, то положення і швидкість завжди є змінними, та складаються з змінної x

та y. Коли ви оновлюєте ці об'єкти гри, вам потрібно обчислити нову позицію

на основі вектора швидкості та пройденого часу. В цій роботі ви побачите, як

це зробити.

Для того, щоб мати можливість використовувати об'єкт ball, потрібно ще кілька спрацьовувань. У методі Game.loadAssets ви завантажуєте червоні, зелені та сині спрайти.Залежно від кольору гармати ви будете змінювати колір м'яча пізніше. Це розширений метод loadAssets :

```
Game.loadAssets = function () {
```

```
var loadSprite = function (sprite) {
return Game.loadSprite("../../assets/Painter/sprites/" + sprite);
};
sprites.background = loadSprite("spr_background.jpg");
sprites.cannon_barrel = loadSprite("spr_cannon_barrel.png");
sprites.cannon_red = loadSprite("spr_cannon_red.png");
sprites.cannon_green = loadSprite("spr_cannon_green.png");
sprites.cannon_blue = loadSprite("spr_cannon_blue.png"); sprites.ball_red
= loadSprite("spr_ball_red.png");
sprites.ball_green = loadSprite("spr_ball_green.png"); sprites.ball_blue =
loadSprite("spr_ball_blue.png");
};
```

Тут ви можете побачити хороший спосіб зробити завантаження спрайтів більш читабельними в JavaScript. Ви оголошуєте локальну змінну loadSprite, яка відноситься до функції. Ця функція приймає в якості параметра ім'я зображення спрайту та викликає метод Game.loadSprite. В якості параметра для цього методу, ви передаєте шлях до папки спрайту плюс ім'я спрайту. Нарешті, функція повертає результат методу Game.loadSprite.

3.2 Створення м'яча

Давайте повернемося до об'єкту ball. У методі initialize цього об'єкта потрібно призначити значення для змінних-членів, як і у випадку з об'єктом cannon. Коли починається гра, м'яч не повинен рухатися. Тому ви ініціалізуєте його швидкість нулем. Крім того, ви спочатку встановили м'яч у нульову позицію. Таким чином, м’яч захований за гарматою, тому, коли м'яч не рухається, ви не бачите його. Ви спочатку встановили колір кульки до як червоний, і ви встановили змінну shooting у стан false. Ось повний метод:

```
ball.initialize = function() {
ball.position = { x : 0, y : 0 };
ball.velocity = { x : 0, y : 0 };
ball.origin = { x : 0, y : 0 }; ball.currentColor = sprites.ball_red;
ball.shooting = false;
};
```

Поряд із методом initialize ви також додаєте метод reset, який скидає позицію м'яча та її статус стрільби:

```
ball.reset = function () {
ball.position = { x : 0, y : 0 };
ball.shooting = false;
};
```

Коли м'яч летить за межі екрану після його вистрілу, ви можете скинути його, викликаючи цей метод. Крім того, до об'єкта ball ви додаєте метод draw. Якщо м'ячем не вистрілили, ви не хочете, щоб гравець його бачив. Отже, ви малюєте спрайт кулі тільки тоді, коли м'ячем вистрілили:

```
ball.draw = function () {
if (!ball.shooting)
return;
Canvas2D.drawImage(ball.currentColor, ball.position, ball.rotation,
ball.origin);
};
```

У тілі цього методу ви можете побачити, що ви використовуєте ключове слово return, щоб намалювати м'яч лише в тому випадку, коли ним не вистрілили. Всередині об'єкта painterGameWorld потрібно викликати методи ігрового циклу для м'яча. Наприклад, це метод draw в painterGameWorld, з якого метод ball.draw викликається:

```
painterGameWorld.draw = function () {
Canvas2D.drawImage(sprites.background, { x : 0, y : 0 }, 0,
{ x : 0, y : 0 });
ball.draw(); cannon.draw();
};
```

Зверніть увагу на порядок, в якому малюються об'єкти гри: спочатку фонове зображення, потім куля, а потім гармата.

3.3 Стрільба м'ячем

Гравець може натиснути ліву кнопку миші на екрані гри, щоб стріляти м'ячем з фарбою. Швидкість м'яча та напрямок, в якій він рухається, визначається позицією, в якій гравець натискає. М'яч повинен рухатися в напрямку цього положення; і чим далі від гармати, тим вище швидкість м'яча. Це інтуїтивно зрозумілий спосіб керування швидкістю м'яча. Для обробки введення ви додаєте метод handleInput до об'єкту ball. Всередині цього методу ви можете перевірити, чи гравець натискає ліву кнопку, використовуючи об'єкт Mouse:

```
if (Mouse.leftPressed)
// do something...
```

Однак, оскільки в будь-який момент в повітрі може бути лише один м'яч, ви можете робити щось, лише якщо м'яч уже не в повітрі. Це означає, що ви

повинні перевірити стан вистрілу м'яча. Якщо м'ячем вже вистрілили, вам не потрібно обробляти натискання миші. Таким чином, ви розширяєте , інструкцію if додатковою умовою, що м'яч в даний час не в повітрі:

```
if (Mouse.leftPressed && !ball.shooting)
// do something...
```

Як ви бачите, ви використовуєте два логічних оператори (&& і !) в сукупності. Через логічний *не* (!)оператор, вся умова в інструкції if буде оцінюватися як true, тільки якщо змінна shooting має значення false : іншими словами м'ячем не вистрілили.

Всередині інструкції if потрібно зробити декілька речей. Ви знаєте, що гравець натиснув десь і що м'яч повинен бути вистрілений. Перше, що вам потрібно зробити, - встановити змінну shooting до правильного значення, тому що статус м'яча потрібно змінити на "вистрілили":

```
ball.shooting = true;
```

Оскільки м'яч зараз рухається, вам потрібно дати *швидкість.* Ця швидкість є вектором у напрямку місця, де гравець натиснув. Ви можете обчислити цей напрямок, витягнувши положення м'яча з позиції миші.

Оскільки швидкість має компоненти х і у, ви повинні зробити це для обох вимірів:

```
ball.velocity.x = (Mouse.position.x - ball.position.x);
ball.velocity.y = (Mouse.position.y - ball.position.y);
```

Обчислення швидкості таким способом також дає бажаний ефект, що, коли користувач натискає далі з гармати, швидкість більша, тоді різниця між положенням миші та положенням кулі також більша. Однак, якщо б ви зараз грали в гру, м'яч буде рухатися трохи повільно. Тому треба помножтит цю швидкість на постійне значення, яке дає м'ячу швидкість, яка підходить для контексту цієї гри:

```
ball.velocity.x = (Mouse.position.x - ball.position.x) * 1.2;
ball.velocity.y = (Mouse.position.y - ball.position.y) * 1.2;
```

Нехай постійне значення буде 1,2. Кожна гра матиме ряд таких *параметрів гри,* які вам потрібно буде налаштувати під час тестування гри, щоб визначити їх оптимальне значення. Знаходження правильних значень для цих параметрів має вирішальне значення для збалансованої гри, і ви повинні переконатися, що вибрані вами значення не роблять гру надто легкою або складною. Наприклад, якщо ви вибрали постійне значення 0,3 замість 1,2, м'яч буде рухатися набагато повільніше.

Якщо додати метод handleInput до ball, він не буде автоматично викликаний. Це потрібно зробити явним об'єктом painterGameWorld. Тому додамо додаткову інструкцію до методу handleInput цього об'єкта:

```
painterGameWorld.handleInput = function () {
ball.handleInput();
cannon.handleInput();
};
```

3.4 Оновлення м'яча

Великою перевагою групування пов'язаних змінних та методів в об'єктах є те, що ви можете тримати кожен об'єкт порівняно невеликим і зрозумілим. Ви можете створювати об'єкти, які більш-менш відображають різні види ігрових об'єктів у грі. У цьому випадку у вас є об'єкт як для гармати, так і для м'яча. Мета полягає в тому, що кожен із цих об'єктів гри стосується вхідних даних гравця, відповідних для цього об'єкта. Ви також хочете, щоб об'єкти гри оновлювалися і малювали себе. Саме тому ви можете розмістити методи update та draw до ball, так що ви можете викликати ці методи в методах ігрового циклу painterGameWorld.

Усередині ball.update потрібно визначити поведінку м'яча. Ця поведінка відрізняється в залежності від того, чи кулька в даний час летить.

Це повний метод:

```
ball.update = function (delta) {
if (ball.shooting) {
ball.velocity.x = ball.velocity.x * 0.99;
ball.velocity.y = ball.velocity.y + 6;
ball.position.x = ball.position.x + ball.velocity.x * delta;
ball.position.y = ball.position.y + ball.velocity.y * delta;
}
else {
if (cannon.currentColor === sprites.cannon_red)
ball.currentColor = sprites.ball_red;
elseif (cannon.currentColor === sprites.cannon_green)
ball.currentColor = sprites.ball_green;
else
ball.currentColor = sprites.ball_blue;
ball.position = cannon.ballPosition();
ball.position.x = ball.position.x - ball.currentColor.width / 2;
ball.position.y = ball.position.y - ball.currentColor.height / 2;
}
if (painterGameWorld.isOutsideWorld(ball.position))
ball.reset();
```

```
};
```

Як ви бачите у заголовку цього методу, він має один параметр, який називається delta. Цей параметр є необхідним, тому що для того, щоб обчислити, якою має бути нова позиція кулі, потрібно знати, скільки часу минуло з часу попереднього виклику update. Цей параметр також може бути корисним в методі handleInput деяких ігрових об'єктів, наприклад, якщо ви хочете дізнатись про швидкість, з якою гравець переміщує мишу, тоді потрібно знати, скільки часу пройшло. Приклад Painter4 розширює кожен об'єкт, який використовує методи ігрового циклу (handleInput , update , draw) таким чином, що час, який минув з моменту останнього оновлення, передавався уздовж параметр Але де ви обчислили значення дельти? І як ви його підраховуєте? У прикладі ви робите це в методі Game.mainLoop :

```
Game.mainLoop = function () {
var delta = 1 / 60;
Game.gameWorld.handleInput(delta);
Game.gameWorld.update(delta);
Canvas2D.clear();
Game.gameWorld.draw();
Mouse.reset();
requestAnimationFrame(Game.mainLoop);
};
```

Оскільки потрібно, щоб цикл гри був виконаний 60 разів за секунду, вирахуйте значення delta:

```
var delta = 1/60;
```

Цей спосіб обчислення минулого часу в циклі гри називається *фіксованим кроком*. Якщо у вас дуже повільний комп'ютер, який не може виконати 60 циклів в секунду ігрового циклу, ви все одно вказуєте свої об'єкти гри, які пройшли лише 1/60 секунди з останнього часу, навіть якщо це може бути неправдою. Як результат, *час гри* відрізняється від *реального часу*. Іншим способом зробити це було б обчислити *фактичний пройденний час*, звернувшись до системного часу. Ось так:

```
var d = new Date();
var n = d.getTime();
```

Змінна n зараз містить кількість мілісекунд з Січня 1, 1970 (!) При кожному запуску гри, ви зберігаете цей час і віднімаете час, який зберігався минулого разу, так ви отримуєте час що пройшов. Такий підхід називается *variable timestep. variable timestep* особливо корисний у іграх, де висока частота кадрів є бажаною. У таких випадках змінна timestep в поєднанні з спробою виклику

методів графічного циклу, якомога частіше, може призвести до більш плавного руху та більш приємного ігрового досвіду.

Недоліком змінних кроків є те, що час триває, навіть якщо гравець тимчасово робить щось інше (наприклад, відкриває меню в грі чи зберігає гру). Як правило, гравці не будуть дуже щасливими, якщо виявлять, що, коли вони переглядали свій інвентар, їхній персонаж був убитий в ігровому світі. Отже, як розробник ігор, ви маєте вирішити такі проблеми, коли використовуєте змінний timestep.

Інший приклад, *variable timestep* може перешкодити іграбельності гри, якщо гравець перемикається в інший додаток (або вкладки в браузері) тимчасово. Це трапляється досить часто, особливо коли ви розробляєте ігри, які запускаються в браузері. Це також є однією з основних причин, через які ви використовуєте фіксований час. Коли гравець переходить на іншу вкладку, виконання коду JavaScript на неактивній вкладці автоматично призупиняється до повернення гравця. При використанні фіксованого кроку, гра просто продовжується з тієї точки, де вона була припинена, коли гравець активує вкладку, тому що ігрові об'єкти не залежать від реального часу, який минув, тільки про фіксованому значенні дельти.

Давайте повернемося до методу ball.update. Якщо ви подивитеся на тіло методу, ви можете побачити, що перша частина складається з інструкції if. Умовою if є те, що змінна ball.shooting повинна мати значення true .Отже, якщо м'яч наразі рухається, виконується тіло інструкції if. Це тіло знову складається з чотирьох інструкцій. Перші дві інструкції оновлюють швидкість, а останні дву оновлюють позицію. Перша інструкція оновлює x в напрямку швидкості. Ви помножуєте швидкість на величину 0,99, ефект якої полягає в тому, що швидкість повільно зменшується.

Це робиться для імітації тертя повітря. Друга команда збільшує швидкість y у кожному оновленні. Це робиться для імітації ефекту, який має назву *гравітація*. Разом, зміна швидкості в напрямках x і y призводить до вірогідності поведінки м'яча. Зрозуміло, в реальному світі гравітація не дорівнює 6. Але тоді знову ж таки ваш реальний світ також не складається з пікселів. Фізика в ігрових світах не завжди точно відображає фізику в реальному світі. Якщо ви хочете включити певну форму фізики у вашу гру, найважливіша частина полягає не в тому, що фізика є реалістичною, а в тому, що *гра є відтворюваною*. Ось чому в стратегічній грі літак буде рухатися так швидко, як воїн, який ходить по землі.

Поточне положення м'яча оновлюється шляхом додавання швидкості до його компонентів x і y.Ось інструкції, які роблять це:

```
ball.position.x = ball.position.x + ball.velocity.x * delta;
ball.position.y = ball.position.y + ball.velocity.y * delta;
```

Як видно, тут використовується змінна delta. Ви обчислюєте нову позицію м'яча на основі швидкості та часу, що пройшов з моменту останнього оновлення. Ви помножуєте кожну величину швидкості з значенням у змінній

delta, і ви додасте результат до поточної позиції м'яча. Таким чином, якщо ви коли-небудь вирішите використовувати більш високу або нижчу частоту кадрів, швидкість руху об'єктів гри не зміниться. Якщо м'яч зараз не в повітрі, ви можете змінити його колір. У цьому випадку треба вибрати поточний колір гармати, змінюючи колір м'яча відповідно. Таким чином, колір м'яча завжди відповідає кольору гармати. Потрібна команда if для обробки різних випадків, як показано нижче:

```
if (cannon.currentColor === sprites.cannon_red)
ball.currentColor = sprites.ball_red;
elseif (cannon.currentColor === sprites.cannon_green)
ball.currentColor = sprites.ball_green;
else
ball.currentColor = sprites.ball_blue;
```

Ви також оновлюєте позицію м'яча:

```
ball.position = cannon.ballPosition ();
ball.position.x = ball.position.x - ball.currentColor.width / 2;
ball.position.y = ball.position.y - ball.currentColor.height / 2;
```

Коли м'яч не знаходиться в повітрі, гравець може змінити свою позицію для пострілу, обертаючи гармату. Тому вам потрібно обчислити правильну позицію м'яча тут, щоб переконатися, що вона відповідає поточній орієнтації гармати. Для цього ви додаєте новий метод, який називається ballPosition, до cannon, в якому ви обчислюєте позицію м'яча.

Використовуючи функції синуса і косинуса, ви обчислите нову позицію таким чином:

```
cannon.ballPosition = function() {
var opp = Math.sin(cannon.rotation) * sprites.cannon_barrel.width * 0.6;
var adj = Math.cos(cannon.rotation) * sprites.cannon_barrel.width * 0.6;
return { x : cannon.position.x + adj, y : cannon.position.y + opp };
};
```

Як видно, ви помножуєте протилежні та сусідні сторони на значення 0,6. Метод повертає новий складний об'єкт, який має змінні x та y, що містять бажані положення x та y м'яча.

Після того, як ви знайдете бажану позицію м'яча, ви віднімете половину ширини і висоти спрайта м’яча від нього. Таким чином, м'яч буде посередині гармати.

Друга частина методу ball.update також є командою if :

```
if (painterGameWorld.isOutsideWorld(ball.position))  ball.reset();
```

Ця частина методу стосується події, яка виникає, коли м'яч виходить за

межі ігрового світу. Щоб розрахувати це правильно, ви додаєте метод, який називається isOutsideWorld до painterGameWorld. Мета цього методу - перевірити, чи є певна позиція поза ігровим світом. Ви визначаєте межі ігрового світу, використовуючи кілька простих правил. Пам'ятайте, що верхній лівий кут екрану є початком. Об'єкт знаходиться за межами ігрового світу, якщо його x-позиція менша за нуль або більше ширини екрана. Об'єкт також знаходиться за межами ігрового світу, якщо його y-позиція перевищує висоту екрана. Якщо ви подивитесь на заголовок цього методу, ви бачите, що він очікує одного параметра, позиції:

```
painterGameWorld.isOutsideWorld = function (position)
```

Якщо ви хочете перевірити, чи є розташування поза екраном, вам треба знати ширину та висоту екрана. У такій HTML5-грі, як Painter, це відповідає розміру *canvas*. Painter4 додає змінну, названу розміром, до гри. Коли метод Game.start визивається, бажаний розмір екрану проходить разом із параметром.Ось розширений метод Game.start :

```
Game.start = function (canvasName, x, y) {
Canvas2D.initialize(canvasName);
Game.size = { x : x, y : y };
Keyboard.initialize();
Mouse.initialize();
Game.loadAssets();
Game.assetLoadingLoop();
};
```

У методі isOutsideWorld ви використовуєте змінну Game.size, щоб визначити, чи знаходиться ця позиція за межами ігрового світу. Тіло методу складається з однієї інструкції, яка використовує return для обчислення логічного значення. *Логічна* операція АБО використовується для покриття різних ситуацій, коли позиція знаходиться за межами ігрового світу:

```
return position.x < 0 || position.x > Game.size.x || position.y > Game.size.y;
```

Повернемося до методу ball.update. Друга команда if викликає метод isOutsideWorld у своєму стані; і якщо цей метод повертає значення true , то виконується метод ball.reset. Або, простіше: якщо м'яч вилетів з екрану, він розміщується на гарматі, готовий знову до стрільби. Тут ви побачите іншу перевагу групувань інструкцій в методах: такі методи, як isOutsideWorld, можуть бути *використані повторно* в різних частинах програми, що дозволяє економити час розробки та призводить до скорочення, більш читабельних програм.

Треба викликати метод ball.update в методі painterGameWorld.update:

```
painterGameWorld.update = function (delta) {
ball.update(delta);
cannon.update(delta);
};
```

Приклад Painter4 дозволяє націлити гармату, вибрати колір, і стріляти м'ячем.

Завдання.

1. Створіть незатухаюче пружне зіткнення, при якому об'єкт «відскакує» від усіх сторін прямокутника.
2. Розширте програму, розроблену в завданні 1: додайте всередину прямокутника - окружність, від якої об'єкт також буде «відскакувати» з урахуванням кута падіння.
3. Розширте програму, розроблену в завданні 2: додайте ще два об'єкти, тепер три об'єкти будуть «відскакувати» від стін і окружності.

Питання для перевірки знань.

1. Чим відрізняється цикл з фіксованим кроком від циклу зі змінним кроком?
2. Коли доцільно застосовувати цикл з фіксованим кроком?
3. Коли доцільно застосовувати цикл зі змінним кроком?
4. Чому час гри відрізняється від реального часу?
5. Чому фізика в ігрових світах не завжди точно відображає фізику в реальному світі?
6. Як додати динамічний об'єкт до ігрового світу?

Лабораторна робота 4
Робота з різними типами об'єктів

Анотація. Лабораторна робота орієнтована на оволодіння студентами навичками створення та керування об'єктами в JavaScript, поняттям класу як засобу для створення декількох ігрових об'єктів певного типу, включення випадковості в ігри.

Мета лабораторної роботи:

- Навчити студентів прийомам створення та керування об'єктами в JavaScript.
- Навчити студентів створювати класи.
- Навчити студентів прийомам включення випадковості в ігри.

У разі успішного виконання лабораторної роботи студент буде вміти створювати та керування об'єктами, створювати класи в ігрових веб-додатках мовою JavaScript, додавати елементи випадковості.

4.1 Розрахунок випадкової швидкості і кольору

Іноді виникають задачі, коли коли треба створити випадкову швидкість та колір для об'єкта, який падає. Для цього можна скористатися методом Math.random. Спочатку подивимося на створення випадкової швидкості. Це можна реалізувати за допомогою окремого методу в класі PaintCan під назвою calculateRandomVelocity . Тут ви використовуєте елемент minVelocity, щоб визначити мінімальну швидкість руху фарб, коли вони падають. Ця змінна дає початкове значення в методі reset, який викликається з конструктора:

```
PaintCan.prototype.reset = function () {
this.moveToTop();
this.minVelocity = 30;
};
```

Ви використовуєте цю мінімальну величину швидкості, коли ви обчислюєте довільну швидкість, яку ви виконуєте в методі calculateRandomVelocity :

```
PaintCan.prototype.calculateRandomVelocity = function () {
return { x : 0, y : Math.random() * 30 + this.minVelocity };
};
```

Метод містить лише одну інструкцію, яка повертає об'єкт, що представляє швидкість.

Швидкість у напрямку x дорівнює нулю, оскільки банки не рухаються горизонтально - вони тільки падають. Швидкість у обчислюється за допомогою генератора випадкових чисел. Щоб отримати позитивне значення швидкості у межах minVelocity і minVelocity + 30, треба помножити це випадкове значення на 30 і додасти значення, що зберігається в змінній minVelocity.

Щоб обчислити випадковий колір, ви також використовуєте генератор випадкових чисел, але ви хочете вибрати з кількох дискретних параметрів (червоний, зелений або синій). Проблема полягає в тому, що Math.random повертає дійсне число від нуля до одиниці. Те, що ви хочете, це генерувати випадкове *ціле* число від 0, 1 або 2. Тоді ви можете використати інструкції if.

Метод Math.floor може допомогти. Math.floor повертає найвище ціле число, яке менше, ніж значення, яке передано як параметр. Наприклад:

```
var a = Math.floor(12.34); // a will contain the value 12
var b = Math.floor(199.9999); // b will contain the value 199
```

```
var c = Math.floor(-3.44); // c will contain the value -4
```

Цей приклад об'єднує Math.random та Math.floor для генерації випадкового числа 0, 1 або 2:

```
var randomval = Math.floor (Math.random () * 3);
```

Використовуючи цей підхід, ви можете обчислити випадкове значення, а потім використати команду if, щоб вибрати колір для фарби. Це завдання виконується методом calculateRandomColor . Ось як виглядає метод:

```
PaintCan.prototype.calculateRandomColor = function () {
var randomval = Math.floor(Math.random() * 3);
if (randomval == 0)
return sprites.can_red; else if (randomval == 1)
return sprites.can_green;
else
return sprites.can_blue;
};
```

Тепер, коли ви запрограмували ці два способи для генерації випадкових значень, ви можете використовувати їх, коли ви визначите поведінку фарби.

4.2 Оновлення Paint Can

Метод update в класі PaintCan має виконувати принаймні наступні речі:

- Встановлювати випадково створену швидкість та колір, якщо банка ще не падає
- Оновлювати позицію банки, додавши швидкість
- Перевіряти, чи пролетіла банка весь шлях.

Для першого завдання ви можете використати команду if, щоб перевірити, чи не рухається банка (швидкість дорівнює нулю). Крім того, введемо трохи непередбачуваності, де з'явиться вона. Щоб досягти цього ефекту, ви призначаєте довільну швидкість та колір лише в тому випадку, якщо якийсь згенерований випадковий номер менше порогу 0,01. Через рівномірний розподіл, лише приблизно в 1 з 100 випадкових чисел число буде менше 0,01. В результаті, тіло інструкції if буде виконуватися тільки іноді, навіть коли швидкість банки дорівнює нулю. У тілі інструкції if ви використовуєте два способи, визначені вами раніше, для генерації випадкової швидкості та випадкового кольору:

```
if (this.velocity.y === 0 && Math.random() < 0.01) {
this.velocity = this.calculateRandomVelocity();
this.currentColor = this.calculateRandomColor();
```

```
}
```

Вам також потрібно оновити позицію банки, додавши поточну швидкість до неї, знову враховуючи час, що минув, як ви зробили з м'ячем:

```
this.position.x = this.position.x + this.velocity.x * delta;
this.position.y = this.position.y + this.velocity.y * delta;
```

Тепер, коли ви ініціалізували банку і оновили її позицію, вам потрібно звернутися до спеціальних випадків. Для фарби можна перевірити, чи не випала вона за межі ігрового світу. Якщо так, то вам потрібно оновити її. Приємно те, що ви вже написали метод, щоб перевірити, чи є певна позиція поза межами ігрового світу: метод isOutsideWorld в класі PainterGameWorld.

Тепер ви можете використовувати цей метод знову, щоб перевірити, чи розташована банка поза межами ігрового світу.

```
if (Game.gameWorld.isOutsideWorld(this.position))
this.moveToTop();
```

Нарешті, щоб зробити гру трохи складнішою, ви трохи збільшите мінімальну швидкість банки кожного разу, коли ви пройдете цикл оновлення:

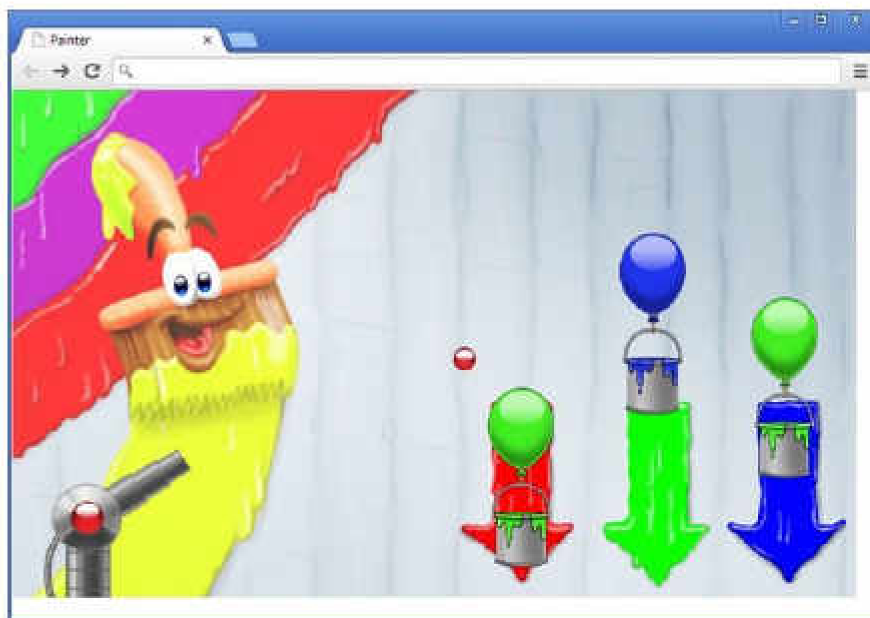
```
this.minVelocity = this.minVelocity + 0.01;
```

Оскільки мінімальна швидкість повільно зростає, гра стає складнішою, коли йде час.

4.3 Малювання банок на екрані

Для того, щоб намалювати банки фарби на екрані, можна додати метод draw в клас PaintCan, який просто малює спрайт банки в потрібному положенні. У класі PainterGameWorld ви викликаєте методи handleInput, update та draw для різних об'єктів гри. Так, наприклад, метод draw в PainterGameWorld виглядає наступним чином :

```
PainterGameWorld.prototype.draw = function ()
{ Canvas2D.drawImage(sprites.background, { x : 0, y : 0 }, 0,
{ x : 0, y : 0 });
this.ball.draw();
this.cannon.draw();
this.can1.draw();
this.can2.draw();
this.can3.draw();
};
```



4-1. *Знімок екрана з прикладом Painter5 з гарматою, м'ячем і трьома банками фарби*

4.4 Представлення позицій і швидкості як векторів

Ви бачили, що класи є цінною концепцією, оскільки вони визначають структуру для об'єктів, а також поведінку для зміни цих об'єктів за допомогою методів. Це особливо корисно, коли вам потрібні декілька подібних об'єктів (наприклад, три банки з фарбою). Інша сфера, де класи можуть бути дуже корисними, полягає у визначенні основних структур даних та методів маніпулювання цими структурами. Спільна структура, яку ви вже бачили, - це об'єкт, який представляє двомірний вектор або вектор швидкості:

```
var position = {x: 0, y: 0};
var anotherPosition = {x: 35, y: 40};
```

На жаль, наступна інструкція не допускається:

```
var sum = position + anotherPosition;
```

Причиною є те, що оператор додавання не визначено для складних об'єктів, таких як ці. Ви, звичайно, можете визначити метод, який виконує цю роботу для вас. Але було б корисно створити ще кілька методів. Наприклад, було б добре, якщо б ви могли відняти такі вектори, помножити їх, обчислити їх довжину тощо. Для того , щоб зробити це, давайте створимо клас Vector2. Почніть з визначення конструктора:

```
function Vector2(x, y) {
this.x = x;
this.y = y;
}
```

Тепер ви можете створити об'єкт наступним чином:

```
var position = new Vector2(0,0);
```

Було б добре, якщо якимось чином ви змогли ініціалізувати вектор без необхідності постійно передавати обидва параметри. Один із способів зробити це - перевірити, чи є x та/або y невизначеними. Якщо це так, ви просто ініціалізуєте змінну як 0, як показано нижче:

```
function Vector2(x, y) {
if (typeof x === 'undefined')
this.x = 0;
else
this.x = x;
if (typeof y === 'undefined')
this.y = 0;
else
this.y = y;
}
```

Ключове слово TypeOf використовується в JavaScript , щоб повернути тип змінної.Тут ви використовуєте це, щоб перевірити, чи мають x і y певний тип. Якщо це так, ви призначаєте значення, передане як параметр, до змінної. В іншому випадку ви призначаєте значення 0. Ось як виглядає той самий метод, але скорочено:

```
function Vector2(x, y) {
this.x = typeof x !== 'undefined' ? x : 0;
this.y = typeof y !== 'undefined' ? y : 0;
}
```

Перед знаком питання є умова. Після знаку запитання є два варіанти значень, розділених двокрапкою. Використовуючи цю коротшу версію, переконайтеся, що ваш код ще читається. Наприклад, ви можете створити об'єкти Vector2 різними способами:

```
var position = new Vector2(); // create a vector (0, 0)
var anotherPosition = new Vector2(35, 40); // create a vector (35, 40)
var yetAnotherPosition = new Vector2(-1); // create a vector (-1, 0)
```

Тепер ви можете додати декілька корисних методів у клас Vector2, щоб було легше виконувати обчислення з векторами. Наприклад, наступний метод робить копію вектора:

```
Vector2.prototype.copy = function () {
return new Vector2(this.x, this.y);
};
```

Це зручно, якщо ви хочете скопіювати позиції або швидкості з різних ігрових об'єктів. Також корисно порівняти вектори. Метод equals робить це для вас:

```
Vector2.prototype.equals = function (obj) {
return this.x === obj.x && this.y === obj.y;
};
```

Ви також можете визначити кілька основних операцій, таких як додавання, віднімання, множення та розділення векторів. Спочатку визначимо метод додавання вектора до існуючого вектора:

```
Vector2.prototype.addTo = function (v) {
this.x = this.x + v.x;
this.y = this.y + v.y; return this;
};
```

Ви можете використовувати цей метод таким чином:

```
var position = new Vector2(10, 10); // create a vector (10, 10)
var anotherPosition = new Vector2(20, 20); // create a vector (20, 20)
position.addTo(anotherPosition); // now represents the vector (30, 30)
```

Остання інструкція методу addTo повертає this. Причиною є те, що існує так званий *ланцюжок операторів*. Оскільки метод addTo повертає вектор в результаті, ви можете викликати методи на цей результат.

Наприклад:

```
var position = new Vector2(10, 10); // create a vector (10, 10)
var anotherPosition = new Vector2(20, 20); // create a vector (20, 20)
position.addTo(anotherPosition).addTo(anotherPosition);
// position now represents the vector (50, 50)
```

Залежно від типу параметра, який передано методу addTo, ви можете зробити щось інше. Якщо параметр - це число, ви просто додасте це число до

кожного елемента вектора. Якщо це вектор, ви виконуєте операцію вже описаним способом. Один із способів це зробити - скористатись оператором typeof, який ви бачили раніше:

```
Vector2.prototype.addTo = function (v) {
if (typeof v === 'Vector2') {
this.x = this.x + v.x;
this.y = this.y + v.y;
}
elseif (typeof v === 'Number') {
this.x = this.x + v;
this.y = this.y + v;
}
return this;
};
```

Ви використовуєте команду if, щоб визначити тип переданого параметра, і ви виконуєте операцію додавання відповідно. Іншим способом визначення типу є використання змінної constructor, яка є частиною кожного об'єкта в JavaScript (так само, як prototype є частиною кожної функції). Це версія методу addTo, яка використовує змінну constructor замість оператору typeof:

```
Vector2.prototype.addTo = function (v) {
if (v.constructor === Vector2) {
this.x = this.x + v.x;
this.y = this.y + v.y;
}
else if (v.constructor === Number) { this.x = this.x + v;
this.y = this.y + v;
}
return this;
};
```

Метод addTo додає вектор до існуючого вектора. Ви також можете визначити метод add, який додає два вектора і повертає *новий вектор*. Для цього ви можете повторно використовувати методи copy та addTo :

```
Vector2.prototype.add = function (v) {
var result = this.copy();
return result.addTo(v);
};
```

Тепер ви можете зробити наступне:

```
var position = new Vector2(10, 10); // create a vector (10, 10)
var anotherPosition = new Vector2(20, 20); // create a vector (20, 20)
var sum = position.add(anotherPosition); //creates a new vector (30, 30)
```

У цьому прикладі position та anotherPosition залишаються незмінними у третій інструкції. Створено новий векторний об'єкт, який містить суму значень у векторних операндах. Подивіться на файл Vector2.js в прикладі Painter6, де ви можете побачити повне визначення класу Vector2. Вона визначає найпоширеніші векторні операції в цьому класі, включаючи методи додавання, описані в цьому розділі. Як результат, використання векторів в грі Painter набагато легше.

Ви використовуєте тип Vector2 у всіх об'єктах гри, щоб відобразити позиції та швидкості.Наприклад, це новий конструктор класу Ball :

```
function Ball() {
this.position = new Vector2();
this.velocity = new Vector2();
this.origin = new Vector2();
this.currentColor = sprites.ball_red;
this.shooting = false;
}
```

Завдяки методам у класі Vector2 ви можете інтуїтивно оновлювати позицію м'яча за своєю швидкістю, в одному рядку коду:

```
this.position.addTo (this.velocity.multiply (delta));
```

4.5 Стандартні значення для параметрів

Ще раз подивимося, як визначено конструктор Vector2 :

```
function Vector2(x, y) {
this.x = typeof x !== 'undefined' ? x : 0;
this.y = typeof y !== 'undefined' ? y : 0;
}
```

Інструкція що малює фонове зображення на екрані:

```
Canvas2D.drawImage (sprites.background, {x: 0, y: 0}, 0, {x: 0, y: 0});
```

Ви можете зробити виклик цього методу набагато компактнішим, дозволяючи методу drawImage автоматично надавати значення за замовчуванням для параметрів позиції, обертання та початку.

```
Canvas2D_Singleton.prototype.drawImage = function (sprite, position, rotation,
```

```
origin) {
    position = typeof position !== 'undefined' ? position : Vector2.zero;
    rotation = typeof rotation !== 'undefined' ? rotation : 0;
    origin = typeof origin !== 'undefined' ? origin : Vector2.zero;
    // remaining drawing code here
    ...
    }
```

Малювання фону робиться наступним чином:

```
Canvas2D.drawImage (sprites.background);
```

Хоча параметри за замовчуванням дуже корисні при створенні компактного коду, переконайтеся, що ви завжди надаєте документацію для своїх методів, які вказують, які значення за замовчуванням використовуватимуться, якщо виклик методу не надає усі параметри.

Завдання.

Розробити гру: хом'якова ферма. Створити клас «хом'як» (англ - "Hamster"). Об'єкти-хом'яки повинні мати масив food для зберігання їжі та метод found для додавання. Додати властивість "рівень ситості".

Питання для перевірки знань.

1. Як оголошувати класи, використовуючи механізм прототипів?
2. Як створити кілька екземплярів типу / классу?
3. Як дододати випадковість до гри?

Лабораторна робота 5
Робота з кольором та підтримка колізій між ігровими об 'єктами

Анотація. Лабораторна робота орієнтована на оволодіння студентами навичками створення властивостей об'єктів, прийомами роботи з кольором.

Мета лабораторної роботи:
- Навчити студентів прийомам створення властивостей об'єктів.
- Навчити студентів прийомами роботи з кольором.

У разі успішного виконання лабораторної роботи студент буде володіти прийомами створення властивостей об'єктів та роботи з кольором.

5.1 Інший шлях представлення кольорів

На даний момент, ви реалізували досить велику частину гри Painter. Ви бачили, як визначити класи ігрових об'єктів за допомогою механізму прототипів. Використовуючи ці класи, ви отримуєте більше контролю над тим, як структуровані ігрові об'єкти та як ви можете створювати ігрові об'єкти певного типу. Ви виділяєте ці визначення класів над різними файлами. Таким чином, коли вам потрібна гармата або м'яч з однаковою поведінкою у майбутній грі, над якою ви працюєте, ви можете просто скопіювати ці файли та створювати екземпляри цих об'єктів гри у вашій грі.

Клас визначає внутрішню структуру об'єкта (з яких змінних він складається), а також методи, які певним чином маніпулюють цим об'єктом.

Ці методи можуть допомогти більш точно визначити, які існують можливості та обмеження об'єкта. Наприклад, якщо хтось хоче повторно використовувати клас Ball, їм не потрібно буде багато детальної інформації про структуру м'яча. Щоб додати літаючий м'яч до гри, достатньо просто створити екземпляр та викликати в грі методи. Як правило, при розробці програми, будь то гра або зовсім інший вид застосування, важливо чітко ігрових веб-додатків» визначити , що можна робити з об'єктами певного класу. Методи - це один із способів зробити це. У цій лабораторній роботі показано ще один спосіб визначити можливості об'єкта: шляхом визначення *властивостей*. Також представлений тип для представлення кольорів та показано, як керувати зіткненням між м'ячем та банкою фарби (якщо це станеться, фарба може змінити колір).

У класі Cannon ви стежили за поточним кольором за допомогою змінної currentColor , яка спочатку вказує на червоний спрайт гармати:

```
this.currentColor = sprites.cannon_red;
```

Ви робили подібне в класі Ball, за винятком того, що ви дозволяли змінній з тим самим ім'ям вказувати на кольорові спрайти м'яча. Хоча це добре працює, це трохи незручно, коли колір м'яча повинен змінюватися відповідно до кольору гармати:

```
if (Game.gameWorld.cannon.currentColor === sprites.cannon_red)
this.currentColor = sprites.ball_red;
else if (Game.gameWorld.cannon.currentColor === sprites.cannon_green)
this.currentColor = sprites.ball_green;
else
this.currentColor = sprites.ball_blue;
```

У інструкції if ви повинні обробляти всі три різні кольори. Крім того, тепер клас Ball повинен мати відомості про спрайти, які використовує клас Cannon. Чи не буде краще, якщо б ви могли більш точно визначати кольори та використовувати це визначення в усіх класах ігрових об'єктів для представлення різних кольорів? Інша причина того, щоб почати об'єднувати

використання кольорів у ваших іграх зараз, полягає в тому, що поточний підхід займає набагато більше часу для програмування, якщо ви коли-небудь вирішите збільшити кількість можливих кольорів у грі (до 4, 6, 10 або більше).

Приклад Painter7 додає файл Color.js . Щоб визначити різні кольори, ви використовуєте підхід, подібний до того, що ви робите, щоб визначити різні клавіші. Цей файл визначає змінну під назвою Color. Змінна "Color" містить ряд підменю, кожне з яких має інший колір.Ви можете визначити кольори таким чином:

```
var Color = {
red : 1,
blue : 2,
yellow : 3,
green : 4,
// and so on
}
```

Використання чисел для подання кольорів не погана ідея, але ви не повинні робити свою власну схему нумерації , яку ніхто не буде знати. Існує вже стандарт для визначення кольору в HTML , який використовує цілі числа, виражені в *шістнадцятковій* формі, і ви можете використовувати той же стандарт. Перевага полягає в тому, що цей підхід широко використовується, широко відомий і широко підтримується засобами (такими як Adobe Kuler).

У стандарті HTML можна визначити кольори елементів на веб-сторінці, використовуючи шістнадцяткове представлення. Наприклад:

```
<body style="background: #0000FF">
That's a very nice background.
</body>
```

У цьому випадку ви вказуєте, що колір фону тіла має бути *синім*. Шістнадцяткове представлення дозволяє визначати кольори у значеннях Червоного, Зеленого, Синього (RGB), де *00* означає, що компонент кольору відсутній, а *FF* означає, що колірна складова є максимальною.

Знак # не є частиною числа, він вказує браузеру, що далі йде шістнадцяткове число, а не десяткове число. Таким чином, шістнадцяткове число #0000FF відображає синій колір, #00FF00 зелений, а #FF0000 червоний.

Ось частина змінної Color:

```
var Color = {
aliceBlue: "#F0F8FF",
antiqueWhite: "#FAEBD7",
aqua: "#00FFFF",
aquamarine: "#7FFFD4",
```

```
azure: "#F0FFFF",
beige: "#F5F5DC",
bisque: "#FFE4C4",
black: "#000000",
blanchedAlmond: "#FFEBCD",
blue: "#0000FF",
blueViolet: "#8A2BE2",
brown: "#A52A2A",
// and so on
}
```

Щоб отримати більш повний перелік кольорів, перегляньте файл Color.js .Тепер ви можете почати використовувати ці визначення кольорів у своїх класах.

5.2 Контроль доступу до даних об'єктів

Три класи ігрових об'єктів представляють об'єкт певного кольору: Cannon , Ball і PaintCan .

Для простоти, почнемо з того, як ви можете змінити клас Cannon, щоб використовувати визначення кольорів з попереднього розділу. До цих пір саме так виглядав конструктор Cannon:

```
function Cannon() {
this.position = new Vector2(72, 405);
this.colorPosition = new Vector2(55, 388);
this.origin = new Vector2(34, 34);
this.currentColor = sprites.cannon_red;
this.rotation = 0;
}
```

Можна додати іншу змінну, яка дає поточний колір гармати. Таким чином, новий конструктор Cannon буде таким:

```
function Cannon() {
this.position = new Vector2(72, 405);
this.colorPosition = new Vector2(55, 388);
this.origin = new Vector2(34, 34);
this.currentColor = sprites.cannon_red;
this.color = Color.red;
this.rotation = 0;
}
```

Однак це не ідеал. Тепер ви зберігаєте надлишкові дані, оскільки інформація про кольори представлена двома змінними. Крім того, можна ввести помилки таким чином,якщо ви забудете змінити одну з двох змінних, коли змінюється колір гармати. Інший спосіб зробити це - не зберігати посилання на поточний спрайт.

```
function Cannon() {
this.position = new Vector2(72, 405);
this.colorPosition = new Vector2(55, 388);
this.origin = new Vector2(34, 34);
this.color = Color.red;
this.rotation = 0;
}
```

Це також не ідеальний підхід, тому що вам потрібно буде шукати правильний спрайт кожен раз, коли ви викликаєте метод draw. Одним з рішень є визначення двох методів, які дозволяють користувачам класу Cannon отримувати інформацію про колір. Потім ви можете залишити конструктор як є, але додати методи для читання і запису значення кольору. Наприклад, ви могли б додати наступні два методи в прототип Cannon:

```
Cannon.prototype.getColor = function () {
if (this.currentColor === sprites.cannon_red)
return Color.red;
else if (this.currentColor === sprites.cannon_green)
return Color.green;
else
return Color.blue;
};
Cannon.prototype.setColor = function (value) {
if (value === Color.red)
this.currentColor = sprites.cannon_red;
else if (value === Color.green)
this.currentColor = sprites.cannon_green;
else if (value === Color.blue)
this.currentColor = sprites.cannon_blue;
};
```

Тепер користувач класу Cannon не повинен знати, що ви використовуєте спрайт, щоб визначити поточний колір гармати. Користувач може просто пройти уздовж визначення кольору, щоб читати або записати його колір:

```
myCannon.setColor(Color.blue);
var cannonColor = myCannon.getColor();
```

Іноді програмісти називають такі методи getters і seters. У багатьох об'єктно-орієнтованих мовах програмування, методи - єдиний спосіб доступу до аних усередині об'єкта, тому для кожної змінної-члена, яка повинна бути доступна поза класом, програмісти додали getters і seters. JavaScript надає функцію, відносно нову для об'єктно-орієнтованих мов програмування: властивості. Властивість замінює getters і seters. Вона визначає, що відбувається, коли ви витягаєте дані з об'єкта і що відбувається, коли ви привласнюєте значення даними всередині .

5.4 Властивість тільки для читання

JavaScript має зручний метод під назвою defineProperty, який дозволяє додавати властивості до класів. Цей метод є частиною об'єкта Object. У об'єкта є ще кілька корисних методів. Метод defineProperty очікує три параметра:

- Прототип, до якого має бути додано властивість (наприклад, Cannon.prototype)
- Ім'я властивості (наприклад, колір)
- Об'єкт, що містить не більше двох змінних: get і set

Кожна змінна get і set вказує на функцію, яка повинна виконуватися при читанні або запису властивості. Однак можна визначити тільки елемент get або set. Це може бути корисно, якщо властивість тільки зчитує інформацію і не може змінювати інформацію. Якщо властивість тільки зчитує інформацію, вона називається властивістю тільки для читання. Ось простий приклад властивості, доступної тільки для читання, яку ви додаєте в клас Cannon:

```
Object.defineProperty(Cannon.prototype, "center",
{
get: function () {
return new Vector2(this.currentColor.width / 2,
this.currentColor.height / 2);
}
});
```

Як ви бачите, ви надаєте три параметра методу defineProperty: прототип, ім'я та об'єкт.

```
var cannonCenter = cannon.center;
```

Аналогічним чином, ви можете додати властивість, яка забезпечує висоту гармати:

```
Object.defineProperty(Cannon.prototype, "height",
{
```

```
get: function () {
return this.currentColor.height;
}
});
```

Навіть можна визначити властивість ballPosition, яка обчислює позицію, в якій має бути куля:

```
Object.defineProperty(Cannon.prototype, "ballPosition",
{
get: function () {
var opposite = Math.sin(this.rotation) * sprites.cannon_barrel.width * 0.6;
var adjacent = Math.cos(this.rotation) * sprites.cannon_barrel.width * 0.6;
return new Vector2(this.position.x + adjacent, this.position.y + opposite);
}
});
```

Так само, як з методами, ви використовуєте це ключове слово для посилання на об'єкт, до якого належить властивість. Приклад Painter7 додає властивості до різних класів. Наприклад, клас Ball також містить властивість центру. У поєднанні зі зручними методами ви додали Vector2, тепер можна розрахувати нове положення кулі в залежності від обертання гармати в одному рядку коду:

```
this.position = Game.gameWorld.cannon.ballPosition.subtractFrom
(this.center);
Object.defineProperty(Vector2, "zero",
{
get: function () {
return new Vector2();
}
});
```

Тепер у вас є дуже короткий спосіб створення двовимірного вектора наступним чином:

```
var position = Vector2.zero;
```

З цього моменту будемо використовувати обидва властивості і методи для визначення поведінки і доступу до даних для об'єктів. Визначаючи корисні властивості і методи в ваших класах, ігровий код стає зазвичай коротше і набагато легше читати. Наприклад, перш ніж у вас були класи з корисними методами і властивостями, так вам потрібно було розрахувати положення м'яча:

```
this.position = Game.gameWorld.cannon.ballPosition();
this.position.x = this.position.x - this.currentColor.width / 2;
this.position.y = this.position.y - this.currentColor.height / 2;
```

5.4 Отримання кольору гармати

У цьому розділі ви визначаєте новий тип «Колір». Тому давайте будемо використовувати цей тип в поєднанні з властивістю читати і писати колір гармати:

```
Object.defineProperty(Cannon.prototype, "color",
{
get: function () {
if (this.currentColor === sprites.cannon_red) return Color.red;
elseif (this.currentColor === sprites.cannon_green) return Color.green;
else
return Color.blue;
}
});
```

Тепер ви можете використовувати цю властивість, щоб отримати доступ до кольору cannon. Наприклад, ви можете зберегти його в змінній:

```
var cannonColor = cannon.Color;
```

Якщо ви хочете привласнити значення cannon color, то для цього вам потрібно визначити задану частину властивості. У цій частині вам необхідно змінити значення змінної currentColor. Це значення надається, коли властивість використовується в іншому методі.

```
cannon.color = Color.Red;
```

Знову ж таки, ви використовуєте інструкцію if, щоб визначити, яке буде нове значення змінної currentColor. Права частина завдання передається заданій частині в якості параметра. Повна властивість потім задається наступним чином:

```
Object.defineProperty(Cannon.prototype, "color",
{
get: function () {;
if (this.currentColor === sprites.cannon_red) return Color.red;
elseif (this.currentColor === sprites.cannon_green) return Color.green;
else
return Color.blue;
```

```
},
set: function (value) {
if (value === Color.red)
this.currentColor = sprites.cannon_red; else if (value === Color.green)
this.currentColor = sprites.cannon_green; else if (value === Color.blue)
this.currentColor = sprites.cannon_blue;
}
});
```

Це приклад властивості, яку можна прочитати і записати. Ви додаєте властивість кольору до всіх кольорових типів ігрових об'єктів: Cannon, Ball і PaintCan. Єдина відмінність в коді в частинах get і set - це спрайти для представлення кольорів. Наприклад, це властивість кольору класу Ball:

```
Object.defineProperty(Ball.prototype, "color",
{
get: function () {
if (this.currentColor === sprites.ball_red) return Color.red;
elseif (this.currentColor === sprites.ball_green) return Color.green;
else
return Color.blue;
},
set: function (value) {
if (value === Color.red) this.currentColor = sprites.ball_red;
else if (value === Color.green) this.currentColor = sprites.ball_green;
else if (value === Color.blue) this.currentColor = sprites.ball_blue;
}
});
```

Оскільки ви визначили ці властивості, тепер ви можете легко змінити колір кульки залежно від colorcannon, в одному рядку коду:

```
this.color = Game.gameWorld.cannon.color;
```

5.5 Обробка зіткнень між Ball і Cans

Об'єкти стикаються, вам потрібно обробити це зіткнення в методі поновлення одного з двох об'єктів. В цьому випадку ви можете обробляти конфлікти в класі Ball або в класі PaintCan. Painter7 обробляє зіткнення в класі PaintCan, тому що якщо ви повинні зробити це в класі Ball, вам потрібно буде повторити один і той же код три рази, один раз для кожної фарби. Керуючи конфліктами в класі PaintCan, ви отримуєте цю поведінка автоматично, тому що кожен може перевірити, стикається він чи ні. Хоча перевірка зіткнень може бути виконана різними способами, ви використовуєте тут дуже простий метод.

Визначаєте, що існує зіткнення між двома об'єктами, якщо відстань між їх центрами менше певного значення. Положення центру Ball в будь-який момент в ігровому світі обчислюється шляхом додавання центру кулі спрайту в положення м'яча. Ви можете розрахувати центр фарби аналогічним чином. Оскільки ви додали кілька корисних властивостей для обчислення центрів ігрових об'єктів, давайте використовувати їх для розрахунку відстані між кулею і фарбою, в такий спосіб:

```
var ball = Game.gameWorld.ball;
var distance = ball.position.add(ball.center).subtractFrom(this.position)
.subtractFrom(this.center);
```

Тепер, коли ви розрахували цей вектор, вам потрібно перевірити, чи буде його довжина як в x- та в y-напрямках менше певного значення. Якщо абсолютне значення x-компонента вектора відстані менше, ніж x-значення центру, це означає, що об'єкт кулі знаходиться в межах x діапазону банки.

Той же принцип справедливий для y-напрямку. Якщо це виконується як для x-, так і для y-компонентів, ви можете сказати, що куля стикається з банкою.

Ви можете написати інструкцію if, яка перевіряє цю умову:

```
if (Math.abs(distance.x) < this.center.x &&
Math.abs(distance.y) < this.center.y) {
// handle the collision
}
this.color = ball.color;
ball.reset ();
```

Як ви, напевно, помітили, метод виявлення зіткнень, що використовується тут, не надто точний.

Завдання.

Чорний прямокутник представляє гравця, який при своєму русі обходить весь екран. Додати жовті квадрати (монети), які рухаються випадковим чином. Порахувати кількість монет, які збере гравець (при зіткненні з монетою).

Питання для перевірки знань.

1. Як додавати властивості до класу?
2. Як підтримувати колізії між ігровими об'єктами?
3. Як створювати ігрові об'єкти, які міють різні кольори?

Лабораторна робота 6
Установка кількості життів в ігрових веб-додатках

Анотація. Лабораторна робота орієнтована на отримання студентами практичних навичок роботи з обмеженою кількістю життів в ігрових веб-додатках мовою JavaScript.

Мета лабораторної роботи:

- Навчити студентів додавати обмежену кількість життів гравця.
- Навчити студентів обробляти цикли мовою JavaScript.

У разі успішного проходження практичного заняття студент буде знати основні прийоми роботи з обмеженою кількістю життів в ігрових веб-додатках мовою програмування JavaScript.

6.1 Підтримка кількості життів

У цій лабораторній роботі ви зробите гру Painter цікавішою, надаючи гравцю обмежену кількість життів. Якщо гравці пропускають забагато консервних банок, вони вмирають. У розділі обговорюється, як боротися з цим і як відобразити поточну кількість живих для гравця. Для того, щоб зробити останнє, ви дізнаєтеся про кілька програмних конструкцій для повторення групи інструкцій кількома разів Щоб уявити якусь небезпеку та стимул працювати в грі, ви хочете обмежити кількість банок з фарби неправильного кольору.

Приклад Painter8 додає подібну поведінку до гри та використовує ліміт у п'ять.

Вибір обмеження на п'ять контейнерів для фарб є одним із багатьох прикладів рішень, які ви повинні зробити як дизайнер і розробник гри. Якщо ви дасте гравцеві лише одне життя, то гра буде занадто важкою. Надання гравцям сотні життів знімає стимул для гравця грати добре.

Визначення розумних значень параметрів часто відбувається під час тестування гри. Окрім тестування самої гри, ви можете також попросити своїх друзів або родичів грати в гру, щоб отримати уявлення про те, які значення слід вибрати для цих параметрів.

Щоб зберегти ліміт життя, ви додасте додаткову змінну-члену до класу PainterGameWorld :

```
this.lives = 5;
```

Ви спочатку встановили це значення в 5 в конструкторі класу PainterGameWorld. Тепер ви можете оновлювати значення кожного разу, коли фарба може випасти поза екраном. Ви виконуєте цю перевірку в методі update класу PaintCan. Тому вам потрібно додати кілька інструкцій у цьому методі. Єдине, що вам потрібно зробити, це перевірити, чи може колір фарби бути

таким самим, як його цільовий колір, коли він падає в нижній частині екрана. Якщо це так, вам доведеться зменшити лічильник lives на PainterGameWorld клас Перш, ніж це зробити, ви повинні розширити клас PaintCan, щоб об'єкти PaintCan знали, що вони повинні мати цільовий колір, коли вони випадають з нижньої частини екрана. Painter8 проходить уздовж цього цільового кольору в якості параметра при створенні об'єктів залили в PainterGameWorld:

```
this.can1 = new PaintCan(450, Color.red);
this.can2 = new PaintCan(575, Color.green);
this.can3 = new PaintCan(700, Color.blue);
```

Ви зберігаєте цільовий колір в змінній в кожній склянці з фарбою:

```
function PaintCan(xPosition, targetColor) {
this.currentColor = sprites.can_red;
this.velocity = Vector2.zero;
this.position = new Vector2(xPosition, -200);
this.origin = Vector2.zero;
this.targetColor = targetColor; this.reset();
}
```

Тепер ви можете розширити метод update PaintCan, щоб він обробляв ситуацію, коли фарба може випасти за межі нижньої частини екрана. Якщо це станеться, вам потрібно перемістити фарбу назад до верхньої частини екрана. Якщо поточний колір фарби може не відповідати цільовому кольору, ви зменшите кількість життів на одне:

```
if (Game.gameWorld.isOutsideWorld(this.position)) {
if (this.color !== this.targetColor)
Game.gameWorld.lives = Game.gameWorld.lives - 1;
this.moveToTop();
}
```

Можливо, вам доведеться скоротити кількість життів більш ніж в одній точці. Щоб полегшити це, ви можете змінити штраф у змінній:

```
var penalty = 1;
if (Game.gameWorld.isOutsideWorld(this.position)) {
if (this.color !== this.targetColor)
Game.gameWorld.lives = Game.gameWorld.lives - penalty;
this.moveToTop();
}
```

Таким чином, ви можете ввести більш жорсткі покарання, якщо хочете, або

динамічні покарання (перша втрата коштує одне життя, друга втрата коштує два, і так далі). Ви також можете уявити, що іноді може падати спеціальна фарба. Якщо гравець стріляє м'ячем правильного кольору, штраф за невідповідність кольору фарби тимчасово стає нулем.

6.2 Показати кількість життів для гравця

Потрібно якось на екрані вказувати, скільки живе гравець. У грі Painter ви це робите, показуючи кілька кульок у верхньому лівому куті екрана. Використовуючи знання, яке ви маєте, ви можете використовувати інструкцію if :

```
if (lives === 5) {
// Draw the balloon sprite 5 times in a row
} else if (lives === 4) {
// Draw the balloon sprite 4 times in a row
} else if (lives === 3)
// And so on...
```

Це не дуже хороше рішення. Це призводить до великої кількості коду, і вам доведеться копіювати ту саму інструкцію багато разів. На щастя, є краще рішення: *ітерація* .

6.3 Виконання інструкцій кілька разів

Ітерація в JavaScript – це спосіб повторити вказівки кілька разів. Подивіться на наступний фрагмент коду:

```
var val = 10;
while (val >=3)
val = val - 3;
```

Друга команда називається циклом. Ця інструкція складається з певного заголовку (в той час як (val> = 3)) і тіла (val = val - 3;), що дуже схоже на структуру if. Заголовок складається з слова while та *умови* в дужках.

Само тіло є інструкцією. У цьому випадку команда віднімає 3 від змінної. Тим не менш, це могло б також бути іншим видом інструкцій, таких як виклик методу або доступ до властивості. Малюнок 6.1 відоюражує синтаксичну діаграму інструкції.

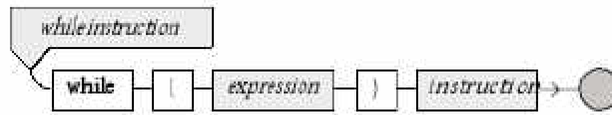


Рис. 6.1. *Синтаксична схема інструкції* while

Коли команда в while виконується, тіло цикла виконується кілька разів. По суті, якщо *умова* в заголовку дає true , тіло буде виконано. Якщо ви хочете намалювати кількість гравців на екрані, ви можете використовувати інструкцію в режимі часу, щоб зробити це досить ефективно:

```
var i = 0;
while (i < numberOfLives) {
Canvas2D.drawImage(sprites.lives,
new Vector2(i * sprites.lives.width + 15, 60)); i = i + 1;
}
```

У цій інструкції while тіло виконується до тих пір, як змінна i містить значення менше ніж numberOfLives. Кожного разу, коли тіло виконується, ви малюєте спрайт на екрані, а потім збільшуєте i на 1. Результатом є те , що ви малюєте спрайт на екрані рівно numberOfLives раз! Отже, ви використовуєте змінну - *лічильник* .

Як ви можете бачити, тіло інструкції в while може містити більше однієї інструкції. Якщо тіло містить більше однієї інструкції, інструкції повинні бути поміщені між фігурними дужками. Позиція, на якій ви малюєте спрайт, залежить від значення i .Таким чином, ви можете малювати кожен спрайт трохи далі вправо, так що вони прекрасно поміщаються в ряд. Коли ви вперше виконуєте тіло, ви малюєте спрайт у положенні 15. У наступній ітерації ви малюєте спрайт в положенні: позиція x = sprites.lives.width + 15 , ітерація після 2 * sprites.lives.width + 15 і так далі. У цьому випадку ви використовуєте лічильник не тільки для того, щоб визначити, як часто ви виконуєте інструкції, а також *змінювати те, що виконують інструкції.*

На малюнку 6.2 показаний знімок екрана гри Painter, де кількість
життів відображається в лівому верхньому кутку.

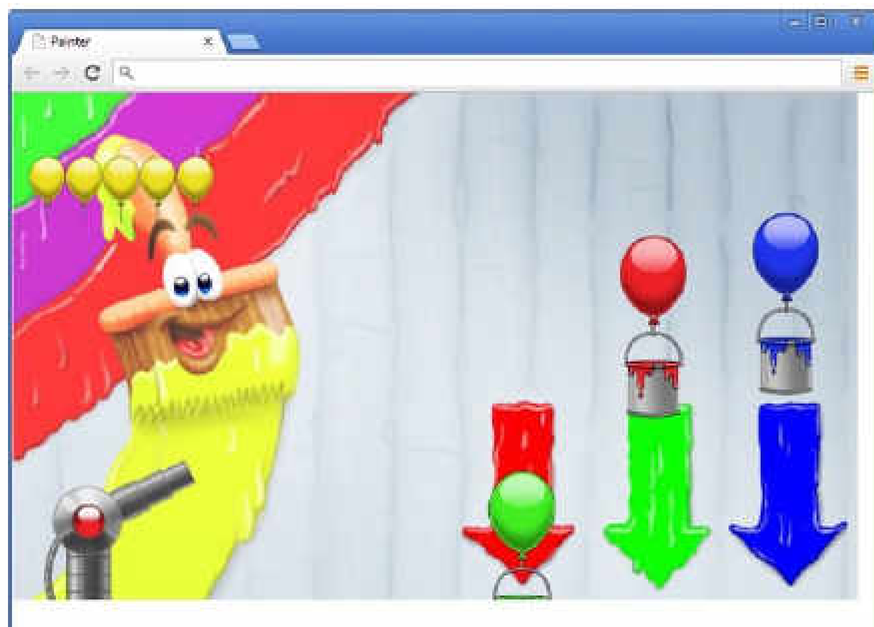


Рис. 6.2. Кількість життів

6.4 Більш компактний циклічний синтаксис

Багато while інструкції використовують змінну підрахунку i, отже, мають наступну структуру:

```
var i;
i = begin value;
while (i < end value ) {
// do something useful using i i++;
}
```

Оскільки цей вид інструкції досить поширений, для нього є більш компактні позначення:

```
var i;
for (i = begin value ; i < end value ; i++ ) {
// do something useful using i
}
```

Сенс цієї інструкції точно такий же, як і раніше. Перевага використання такої інструкції в даному випадку - це те , що треба зробити з лічильником, красиво згруповано в заголовку інструкції. Це зменшує ймовірність того, що ви забудете збільшити лічильник (в результаті чого ви отримаєте нескінченний цикл). Наприклад, подивіться на наступний фрагмент:

```
for (var i = 0; i < this.lives; i++) { Canvas2D.drawImage(sprites.lives,
```

```
new Vector2(i * sprites.lives.width + 15, 60));
}
```

Це дуже компактна інструкція, що збільшує лічильник і відмальовує спрайт в різних позиціях. Ця інструкція еквівалентна наступній інструкції:

```
var i = 0;
while (i < this.lives) {
Canvas2D.drawImage(sprites.lives,
new Vector2(i * sprites.lives.width + 15, 60));
i = i + 1;
}
```

І ось ще один приклад:

```
for (var i = this.lives - 1; i >=0; i--)
Canvas2D.drawImage(sprites.lives,
new Vector2(i * sprites.lives.width + 15, 60));
```

Малюнок 6.3 містить синтаксичну схему інструкції for.

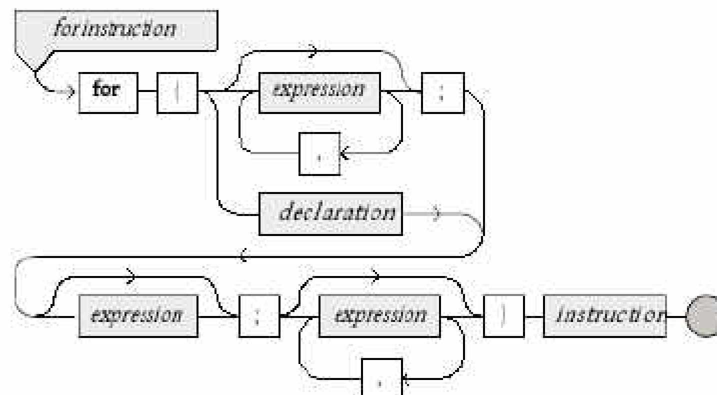


Рис. 6.3. *Синтаксична схема інструкції* for

Є кілька особливих випадків, про які ви повинні знати, коли ви маєте справу з while та for циклів. У наступних підрозділах обговорюються ці випадки.

6.5 Спеціальні випадки

Іноді умова в заголовку інструкції в while вже дорівнює false на самому початку. Подивіться на наступний фрагмент коду:

```
var x = 1;
var y = 0; while (x < y)
x++;
```

В цьому випадку тіло команди в while не виконується, навіть не один раз! Тому в цьому прикладі змінна x зберігає значення 1.

Завдання.

Гравець має обмежену кількість життів. При загибелі гравця віднімається по одному життю. Коли життя закінчуються, гра починається заново. Підредактуйте runGame, щоб вона підтримувала житті. Нехай гравець починає з трьох.

```
<link rel="stylesheet" href="css/game.css">
<body>
<script>
// Стара функція runGame – підредактуйте її...
function runGame(plans, Display) {
function startLevel(n) {
runLevel(new Level(plans[n]), Display, function(status) {
if (status == "lost")
startLevel(n);
else if (n < plans.length - 1)
startLevel(n + 1);
else
console.log("You win!");
});
}
startLevel(0);
}
runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

Питання для перевірки знань.

1. Як зберігати та відображати кількість життів шравця?
2. Як повторювати групу інструкцій з використанням інструкцій while та for?
3. Як перезапустити гру, коли у гравця не залишилось життів?

ПРАКТИЧНІ ЗАНЯТТЯ

Практичне заняття 1
Розробка першого веб-додатка

Анотація. Практичне заняття орієнтовано на ознайомлення студентів з побудовою простого ігрового веб-додатка с використанням тега HTML canvas та елементів мови JavaScript.

Мета практичного заняття:

- Відновити в пам'яті основні елементи структури HTML-документу.
- Створити простий веб-додаток з додаванням HTML-тега canvas, який дозволяє додавати графічні елементи в документ.
- Навчитися виносити JavaScript-код в окремий файл.

У разі успішного проходження практичного заняття студент буде знати структуру HTML-документу, способи додавання JavaScript-кода в код HTML, вміти створювати простий веб-додаток з додаванням HTML-тега canvas, який дозволяє додавати графічні елементи в документ.

Створення першого веб-додатку

У цьому розділі ви створюєте кілька дуже простих прикладів програм, що використовують JavaScript. Раніше в розділі ви побачили базову HTML-сторінку:

```
<!DOCTYPE html>
<html>
<head>
<title> Корисний веб-сайт </ title>
</ head>
<body>
Це дуже корисний веб-сайт.
</ body>
</ html>
```

Відкрийте програму редагування тексту, таку як Блокнот, і скопіюйте-вставте цей текст до нього. Збережіть файл як щось з розширенням .html .Потім двічі клацніть цей файл, щоб відкрити його в браузері. Ви бачите майже порожній HTML-сторінку, як показано на малюнку 1-2.

У HTML теги використовуються для структури інформації в документі. Ви можете розпізнати ці теги, оскільки вони поміщені між кутовими дужками. Між такими тегами розміщується кожен різний тип вмісту. Ви можете відрізнити початковий тег від закриваючого тегу, перевіряючи, чи є коса риска перед

іменем тегу.

Наприклад, заголовок документа поміщається між початковим тегом <title> та закриттям тег </ title> .Сам заголовок, у свою чергу, є частиною *заголовку* , який розділений <head> і </ head>. Заголовок міститься в *HTML*- частині, яка розділена тегами <html> та </ html> .Як ви можете бачити, система позначення HTML дозволяє логічно організовувати вміст документа. Загальний HTML-документ має своєрідну структуру дерева, де html- елемент є коренем дерева; корінь складається з таких елементів, як голова і тіло , які в свою чергу складаються з більшості гілок.

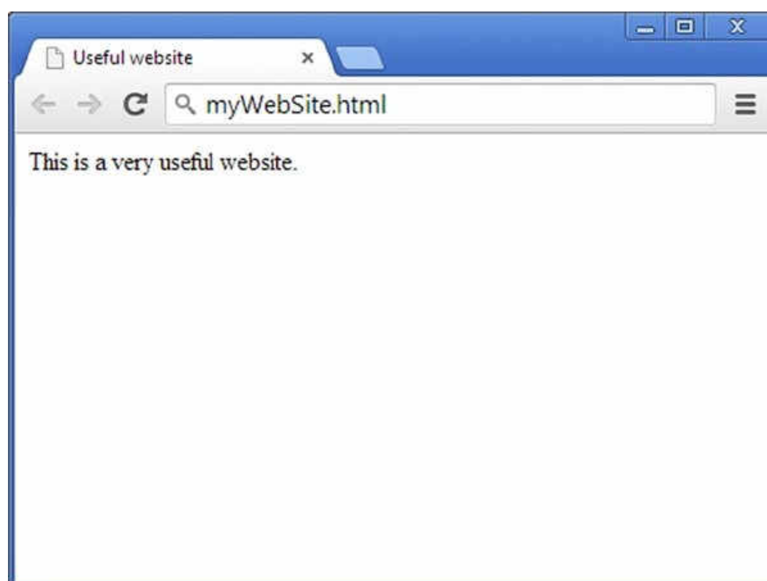


Рисунок 1.1 - *Дуже проста HTML-сторінка*

Створивши документ HTML, ви зможете застосувати *стиль* до нього.Наприклад, вам може знадобитися змінити макет різних частин документа HTML, або ви можете використовувати інший шрифт або застосувати фоновий колір.Стиль можна визначити як частину документа HTML, або ви можете визначити стиль, використовуючи CSS (Каскадні таблиці стилів) файл.

Наприклад, ця проста таблиця стилів визначає поля HTML-сторінки та її тіло до 0:

```
html, body {
маржа: 0;
}
```

Якщо ви хочете, щоб у вашій HTML-сторінці використовувався CSS-файл (таблиця стилів), ви просто додаєте наступний рядок в частину <head> :

```
<link rel = "stylesheet" type = "text / css" href = "game-layout.css" />
```

Я буду використовувати попередню таблицю стилів в більшості прикладів гри в цій книзі. У розділі 13 я розширюю таблицю стилів, щоб мати можливість автоматичного масштабування та позиціонування вмісту на різних пристроях.

Ви також можете змінити стиль у самому документі HTML, а не використовувати CSS-файли для визначення стилю.Це робиться, встановлюючи *атрибути* тегу.Наприклад, тіло наступної сторінки HTML має стиль тегу атрибуту, який призначений для зміни фонового кольору на синій (див. Малюнок 1.2 для сторінки, яка відображається):

```
<0,100DOCTYPE html>
<html>
<head>
<title> BasicExample </ title> </ head>
<body style = "background: blue">
Це дуже хороший фон.
</ body>
</ html>
```

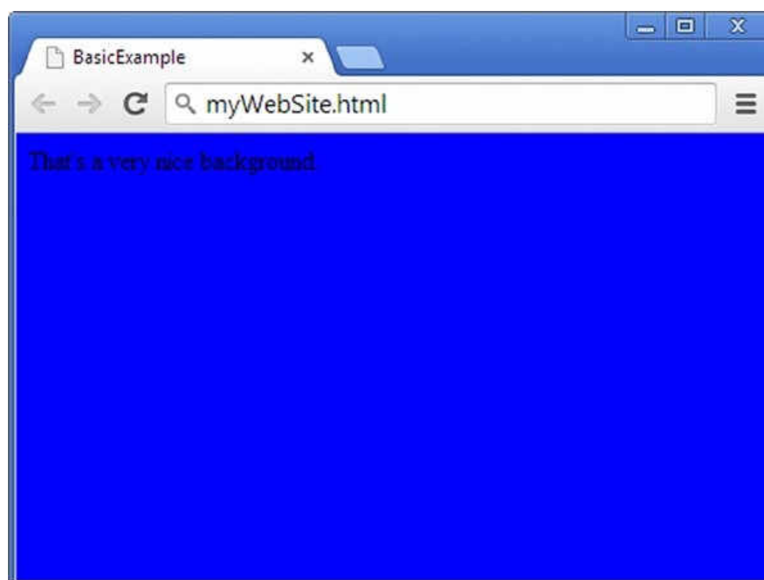


Рисунок 1.2 - *Проста веб-сторінка з синім фоном*

Ви можете змінити різні аспекти стилю за допомогою атрибута style , як в прикладі.Наприклад, розглянемо наступний HTML - документ:

```
<!DOCTYPE html>
<html>
<head>
<title> BasicExample </ title> </ head>
<body>
<div style = "background: blue; font-size: 40px;"> Привіт, як ти?</ div>
```

```
<div style = "background: yellow; font-size: 20px;"> Я роблю великий, дякую!</ div>
</ body>
</ html>
```

Якщо ви подивитеся на вміст body , ви побачите, що він містить дві частини.Кожна частина укладена в DIV - теги, які використовуються для поділу HTML документа на *розділи.*Ви можете застосувати інший стиль до кожного розділу.У цьому прикладі перший розділ має синій фон та розмір шрифту 40 пікселів, а другий розділ має жовтий фон та розмір шрифту 20 пікселів.

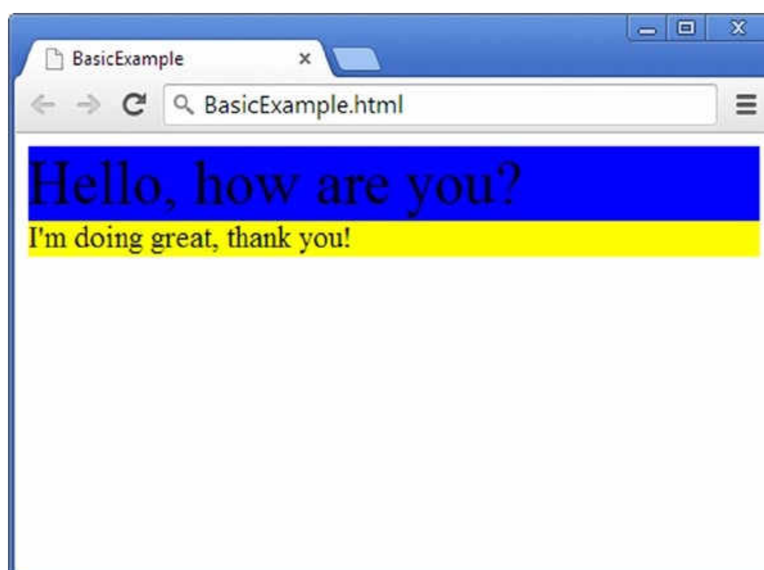


Рисунок 1.3 - *Веб-сторінка, що складається з двох підрозділів, кожна з яких має інший фон та розмір шрифту*

Замість того, щоб додати атрибут style до елемента HTML, ви також можете використовувати JavaScript для зміни стилю цього елемента.Наприклад, ви можете змінити фоновий колір тіла за допомогою JavaScript, як показано нижче:

```
<!DOCTYPE html>
<html>
<head>
<title> BasicExample </ title>
 <script> changeBackgroundColor = function () {
document.body.style.background = "blue";
}
document.addEventListener ('DOMContentLoaded', changeBackgroundColor);
</script>
</ head>
<body>
Дуже гарне ім'я.
```

```
</ body>
</ html>
```

Сторінка, яку показує браузер, виглядає точно так само, як і в першому прикладі (показано на малюнку 1-2), але існує принципова відмінність між використанням JavaScript для цього, а не додавання атрибуту тегу body : зміни коліру JavaScript *динамічно* за допомогою скрипта.Це відбувається тому, що скрипт містить таку лінію:

```
document.addEventListener ('DOMContentLoaded', changeBackgroundColor);
```

Всередині програми JavaScript ви маєте доступ до всіх елементів на сторінці HTML. І коли все станеться, ви можете наказати браузеру виконати інструкції. Тут ви вказуєте, що функція changeBackgroundColor повинна виконуватися після закінчення сторінки завантаження

Існує багато різних типів цих подій у HTML та JavaScript. Наприклад, ви можете додати кнопку до документа HTML та виконувати інструкції JavaScript, коли користувач натискає кнопку. Ось документ HTML, який ілюструє це (див. також Рисунок 1-5):

```
<!DOCTYPE html>
<html>
<head>
<title> BasicExample </ title>
</script>
sayHello = function () {alert ("Hello  World!");
}
document.addEventListener ("click", sayHello);
</script>
</ head>
<body>
<button> Натисніть  мені </ button>
</ body>
</ html>
```

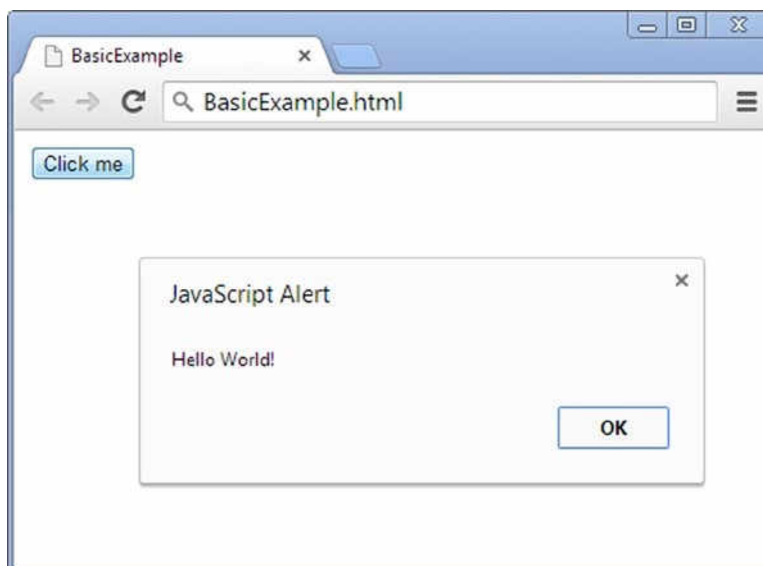


Рисунок 1.4 – Screen shot: *HTML-сторінка, що містить кнопку. Коли користувач натискає кнопку, з'являється сповіщення*

Такий вид динамічної взаємодії можливий, оскільки браузер може виконувати код JavaScript.Якщо ви хочете запрограмувати ігри, ви можете визначити, як гравець повинен взаємодіяти з грою.

HTML5 Canvas

Хороша ідея щодо нового стандарту HTML полягає в тому, що вона містить кілька тегів, які роблять HTML-документи набагато гнучкішими. Дуже важливим тегом, який був доданий до стандарту, є тег canvas , який дозволяє вам малювати 2D та 3D-графіку у HTML-документі.Ось простий приклад:

```
<!DOCTYPE html>
<html>
<head>
<title> BasicExample </ title>
</ head>
<body>
<div id = "gameArea">
<canvas id = "mycanvas" width = "800" height = "480"> </ canvas>
</ div>
</ body>
</ html>
```

Тут ви можете побачити, що тіло містить розділ під назвою gameArea. Всередині цього розділу є елемент canvas, який має ряд атрибутів. Він має ідентифікатор (mycanvas), і має ширину та висоту.Ви можете знову змінити речі в цьому елементі canvas , використовуючи JavaScript. Наприклад, наступний код змінює фоновий колір елемента canvas за допомогою декількох інструкцій JavaScript:

```
<!DOCTYPE html>
<html>
<head>
<title> BasicExample </ title>
</script>
changeCanvasColor = function () {
var canvas = document.getElementById ("mycanvas"); var context = canvas.getContext ("2d"); context.fillStyle = "синій";
context.fillRect (0, 0, canvas.width, canvas.height);
}
document.addEventListener ('DOMContentLoaded', changeCanvasColor);
</script>
</ head>
<body>
<div id = "gameArea">
<canvas id = "mycanvas" width = "800" height = "480"> </ canvas>
</ div>
</ body>
</ html>
```

У функції changeCanvasColor ви вперше знайдете елемент canvas .Це елемент документа HTML, на якому ви можете малювати 2D та 3D-графіку. Використання цього елемента у вашому коді є корисним, оскільки тоді ви легко можете отримати інформацію про полотно, наприклад, її ширину або висоту. Для виконання операцій на полотні (наприклад, накреслення на ньому) потрібен *контекст* полотна .Контейнер полотна надає функції для малювання на полотні. Коли ви отримуєте контекст полотна, потрібно вказати, чи хочете ви малювати 2 або 3 розміри. У цьому прикладі ви отримаєте двомірний контекст полотна. Ви використовуєте його для вибору фонового кольору наповнення та заповнення полотна цим кольором.На малюнку 1-6 показана результуюча сторінка HTML, що відображається браузером.У наступних розділах детально описується елемент canvas і як він використовується для створення ігри

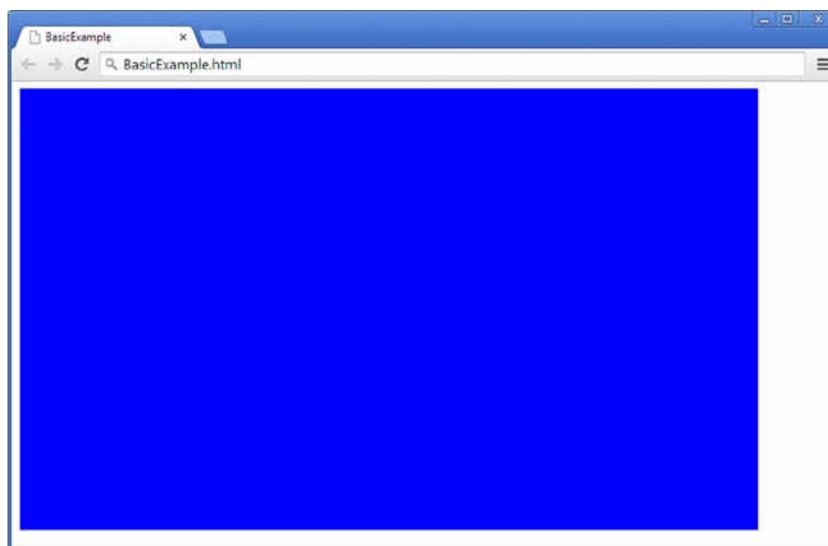


Рисунок 1.5. *- Відображення HTML5 полотна на веб-сторінці та заповнення його кольором*

JavaScript в окремому файлі

Замість написання всього коду JavaScript у документі HTML, ви також можете написати код JavaScript у окремому файлі та включити цей файл у документ HTML:

```
<!DOCTYPE html>
<html>
<head>
<title> BasicExample </ title>
<script src = "BasicExample.js"> </ script>
</ head>
<body>
<div id = "gameArea">
<canvas id = "mycanvas" width = "800" height = "480"> </ canvas>
</ div>
</ body>
</ html>
```

Файл JavaScript BasicExample.js потім містить такий код:

```
changeCanvasColor = function () {
var canvas = document.getElementById ("mycanvas"); var context = canvas.getContext ("2d"); context.fillStyle = "синій";
context.fillRect (0, 0, canvas.width, canvas.height)
}
document.addEventListener ('DOMContentLoaded', changeCanvasColor);
```

У багатьох випадках бажано це зробити. Розділяючи код сценарію з HTML-документа, набагато простіше знайти код або використовувати його на

різних веб-сайтах. Всі приклади, використані в цій книзі, мають код JavaScript, розділений з HTML-документа, добре організований у одному або декількох файлах JavaScript.

Практичне заняття 2
Розміщення, завантаження спрайтів та анімація мовою програмування JavaScript

Анотація. Практичне заняття орієнтовано на отримання студентами практичних навичок розміщення, завантаження спрайтів, та створення динамічних спрайтів мовою JavaScript.

Мета практичного заняття:

- Навчити студентів завантажувати спрайти з файлів.
- Навчити студентів розміщувати спрайти на екрані.
- Навчити студентів створювати ігровий цикл для переміщення спрайтів.

У разі успішного проходження практичного заняття студент буде знати основні прийоми роботи зі спрайтами мовою програмування JavaScript.

Звук - це ще один тип ігрового активу. Він обробляється дуже подібно до спрайтів. Отже, в кінці цього розділу ви також побачите, як відтворювати музичні та звукові ефекти у вашій грі.

Пошук спрайтів

Перш ніж програма може використовувати будь-які активи, вона повинна знати, де шукати ці активи. За замовчуванням браузер, який виконує роль інтерпретатора, шукає спрайт у тій самій папці, що й файл JavaScript. Подивіться на приклад SpriteDrawing, що належить до цієї глави. Ви бачите файл spr_balloon.png у тій самій папці, що й файл HTML та файл JavaScript.Ви можете завантажити цей спрайт і намалювати його на екрані.

Завантаження Sprites

Давайте розглянемо, як можна завантажити спрайт з файлу. Як тільки ви це зробите, ви зберігатимете його десь у пам'яті за допомогою змінної. Вам потрібна ця змінна у кількох різних методах ігрового циклу. У методі start ви завантажуєте спрайт і зберігаєте його в змінній.У методі draw ви можете отримати доступ до змінної, щоб намалювати спрайт на екрані.Тому ви додасте змінну під назвою balloonSprite до об'єкта Game .Тут ви можете побачити оголошення змінної game та його ініціалізації:

```
var Game = {
```

```
canvas : undefined, canvasContext : undefined, balloonSprite : undefined
};
```

У методі start , ви призначаєте значення для цих змінних.Ви вже бачили , як отримати полотно і контекст полотна.Так само , як game, полотно і контекст полотна є *об'єкти,* кожен складаються від інший змінні (або об'єкти). Якщо ви навантаження а спрайт ви мати *ан об'єкт це являє собою спрайт.* ви міг визначити ан об'єкт змінна це містить все в інформація в в зображення:

```
Game.balloonSprite = {
src : "spr_balloon.png", width : 35,
height : 63,
...
}
```

Це стає проблематичним, якщо ви хочете завантажити сотні спрайтів для вашої гри. Кожного разу, ви повинні визначити такий об'єкт, використовуючи літературний об'єкт. Крім того, вам доведеться переконатися, що ви випадково не використовуєте інші імена змінних в об'єкті, оскільки тоді ви матимете непослідовне представлення зображень. На щастя, ви можете уникнути цієї проблеми, використовуючи *типи* .

Тип в основному є визначенням того, що повинен виглядати об'єкт такого типу; це *проект* для об'єкта.Наприклад, JavaScript знає тип, який називається Image .Цей тип вказує, що об'єкт зображення має мати ширину, висоту, вихідний файл тощо. Існує дуже простий спосіб створити об'єкт, який має Тип зображення , використовуючи нове ключове слово:

```
Game.balloonSprite = new Image();
```

Це набагато простіше, ніж потрібно вводити весь зміст, який має мати.Вираз нове new Image() в основному це працює для вас.Використовуючи типи, у вас тепер є простий спосіб створити об'єкти, і ви можете бути впевнені, що ці об'єкти завжди мають однакову структуру.Коли є об'єкт

побудовано , який має структуру, визначається типом Image, ви говорите , що цей об'єкт *має тип* зображення.

Ви ще не вказали, які *дані* слід містити в цій змінній.Ви можете встановити вихідний файл цього зображення, присвоюючи ім'я файлу змінній src, яка завжди є частиною об'єкта Image :

```
Game.balloonSprite.src = "spr_balloon.png";
```

Після встановлення змінної src браузер починає завантаження файлу.Браузер автоматично заповнює дані для змінних ширини та висоти, тому

що він може витягти цю інформацію з вихідного файлу.

Іноді завантаження вихідного файлу займає деякий час. Наприклад, файл може бути збережений на веб-сайті з іншого боку світу. Це означає, що якщо ви спробуєте намалювати зображення одразу після встановлення вихідного файлу, ви можете зіткнутися з проблемами. В результаті, ви повинні переконатися, що кожне зображення завантажується, перш ніж ви можете почати гру. Це дуже акуратно, використовуючи функцію *обробника подій* .У розділі 7 ви бачите, як це працює. На даний момент, просто припустимо, що завантаження зображення не займе більше половини секунди. За допомогою методу setTimeOut ви викликаєте метод mainLoop після затримки 500 мілісекунд:

```
window.setTimeout (Game.mainLoop, 500);
```

Це завершує метод start , який зараз виглядає так:

```
Game.start = function () {
Game.canvas = document.getElementById("myCanvas"); Game.canvasContext = Game.canvas.getContext("2d"); Game.balloonSprite = new Image(); Game.balloonSprite.src = "spr_balloon.png"; window.setTimeout(Game.mainLoop, 500);
};
```

Sprites можна завантажити з будь-якого місця.Якщо ви розробляєте гру в JavaScript, то це хороша ідея , щоб думати про організацію ваших спрайтів.Наприклад, ви могли б поставити всі спрайт, що належать до вашої грі в папці *спрайтів.*Потім ви повинні встановити вихідний файл, наступним чином:

```
Game.balloonSprite.src = "sprites/spr_balloon.png";
```

Або, можливо, ви навіть не використовуєте свої власні зображення, але ви посилаєтеся на зображення, які ви знайшли на іншому веб-сайті:

```
Game.balloonSprite.src = "http://www.somewebsite.com/images/spr_balloon.png";
```

JavaScript дозволяє завантажувати файли зображень з будь-якого місця, яке ви бажаєте. Просто переконайтеся, що під час завантаження зображень з іншого веб-сайту виправлено розташування файлів зображень. В іншому випадку, якщо адміністратор цього веб-сайту вирішить перемістити все, не повідомивши вас, ваша гра не буде працювати більше

Малювання Спрайтів

Завантаження спрайту та збереження його в пам'яті не означає, що спрайт намальований на екрані. Щоб це сталося, потрібно зробити щось у методі draw .Дещо намалювати спрайт на холсті, ви використовуєте метод drawImage, який є

частиною контекстного об'єкта полотна.У JavaScript, коли зображення малюється в певній позиції, ця позиція завжди відноситься до *верхнього лівого кута* зображення.

Ось інструкція, яка малює спрайт у верхньому лівому куті екрана:

```
Game.canvasContext.drawImage(sprite, 0, 0, sprite.width, sprite.height,
0, 0, sprite.width, sprite.height);
```

Метод drawImage має ряд різних параметрів.Наприклад, ви можете вказати, в якій позиції ви хочете намалювати спрайт, чи слід малювати лише частину справа. Ви можете просто називати цей метод і зробити це з цим.Однак, якщо ви думаєте про майбутні ігри, які ви хочете побудувати, ви можете використати *стан малюнка,* щоб намалювати спрайт

Стан рисунка в основному являє собою набір параметрів і перетворень, які будуть застосовані до всіх речей, намальованих у цьому стані. Перевага використання стану малюнка замість окремого виклику методу drawImage полягає в тому, що ви можете виконувати більш складні перетворення з справами.Наприклад, за допомогою контурів малювання ви можете обертати або масштабувати справах, що є дуже корисною функцією в іграх. Створення нового режиму малювання здійснюється за допомогою методу save:

```
Game.canvasContext.save();
```

Потім ви можете застосувати різноманітні перетворення в цьому режимі креслення. Наприклад, ви можете перемістити (або *перекласти*) спрайт на певну позицію:

```
Game.canvasContext.translate(100, 100);
```

Якщо ви зараз називаєте метод drawImage , спрайт втягується в позицію (100, 100).І, як тільки ви закінчите малювати, ви можете видалити стан креслення наступним чином:

```
Game.canvasContext.restore();
```

Для зручності давайте визначимо метод, який робить все це для вас:

```
Game.drawImage = function (sprite, position) { Game.canvasContext.save();
Game.canvasContext.translate(position.x, position.y);
Game.canvasContext.drawImage(sprite, 0, 0, sprite.width, sprite.height,
0, 0, sprite.width, sprite.height); Game.canvasContext.restore();
};
```

Як ви можете бачити, переглянувши *параметри* , цей метод вимагає двох частин інформації: спрайт, який слід намалювати, і позиція, на якій вона повинна бути намальована.Спрайт повинен мати тип зображення (хоча ви не

можу легко примусове виконання це в JavaScript коли ви визначити а функція) The позиція є об'єкт змінна що складається від ан x частина і а y частина Коли ви дзвонити це метод ви мати до забезпечити це інформація Для наприклад, ви може малювати в повітряна куля спрайт при позиція (100, 100) як наступним чином:

```
Game.drawImage(Game.balloonSprite, { x : 100, y : 100 });
```

Ви використовуєте фігурні дужки для визначення літературного об'єкта, який містить компоненти x і y .Як видно, в інструкції, яка викликає метод, дозволено визначати об'єкт. Крім того, ви можете спочатку визначити об'єкт, зберегти його в змінній, а потім за допомогою цієї змінної викликати метод drawImage :

```
var balloonPos = { x : 100,
y : 100
};
Game.drawImage(Game.balloonSprite, balloonPos);
```

Цей код робить точно так само, як попередній виклик drawImage , за винятком того, що набагато довше його писати.Ви можете просто застосувати виклик методу drawImage методом draw , а куля буде намальовано у потрібній позиції:

```
Game.draw = function () {
Game.drawImage(Game.balloonSprite, { x : 100, y : 100 });
};
```

На малюнку 2.1 показано, як вигляд програми виглядає у веб-переглядачі.

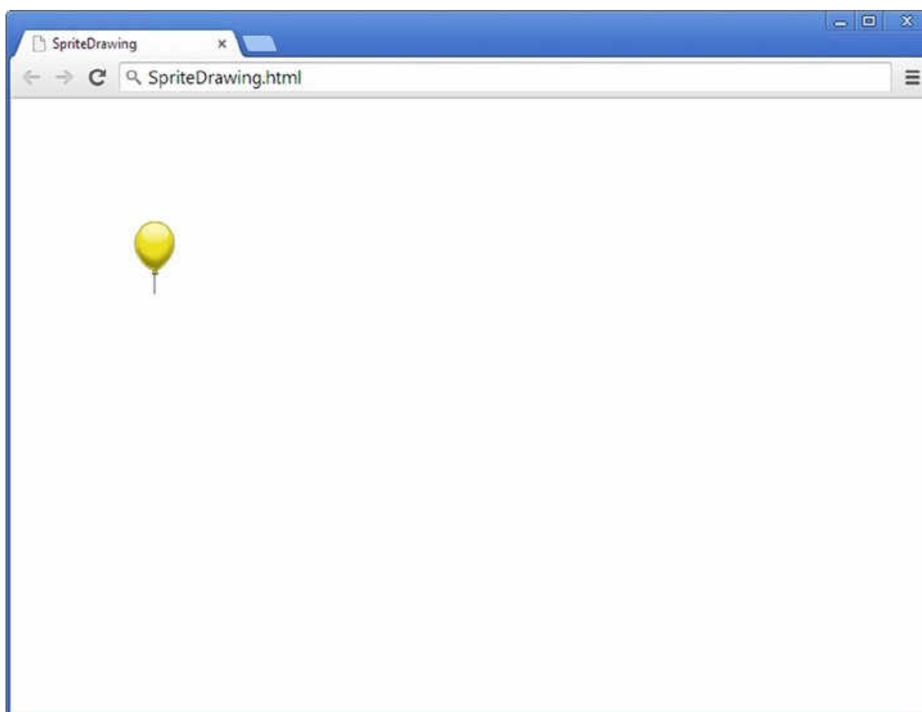


Рисунок 2.1 – *SpriteDrawing*

Знову ж таки, зверніть увагу, що якщо ви повідомляєте браузеру про те, що в певній позиції вказує спрайт, там звертається *верхня ліва* частина спрайту.

Переміщення Спрайтів

Тепер, коли ви можете намалювати спрайт на екрані, ви можете використовувати цикл гри, щоб перемістити його, як і в квадраті в прикладі MovingSquare у розділі 3. Давайте зробимо невелике розширення цієї програми, що змінює позицію балону на основі пройденого часу.Щоб це зробити, вам доведеться десь зберігати розташування кульок. Вам потрібно обчислити цю позицію в способі update і зробити повітряну кулю в цьому положенні в методі draw.Тому ви додаєте змінну до Game об'єкта, що являє собою позицію, як наступним чином:

```
var Game = {
canvas : undefined, canvasContext : undefined, balloonSprite : undefined,
balloonPosition : { x : 0, y : 50 }
};
```

Як видно, ви визначаєте позицію як об'єкт, що складається з двох змінних (x і y) в об'єкті game .Тепер ви можете додати інструкцію до методу update, який змінює x-позицію залежно від часу, як і в прикладі MovingSquare.Ось метод update:

```
Game.update = function () { var d = new Date();
Game.balloonPosition.x = d.getTime() % Game.canvas.width;
```

```
};
```

Тепер залишається лише переконатися, що ви використовуєте змінну balloonPosition під час намалювання куля на екрані методом draw :

```
Game.drawImage(Game.balloonSprite, Game.balloonPosition);
```

Завантаження та малювання декількох спрайтів

Створення ігор з лише рівнинним білим фоном є дещо нудним.Ви можете зробити свою гру а трохи більше візуально привабливий по відображення а фон спрайт Це засоби ви мати до навантаження інший спрайт в способі запуску і продовжити метод малювання, щоб намалювати його.Остаточна версія цієї програми називається FlyingSprite, і ви можете знайти повний вихідний код у теці зразків, що належать до цієї глави.Якщо ви відкриєте програму FlyingSprite в вашому браузері, ви бачите , що тепер два спрайт намальовані: фон і, поверх нього, повітряна куля.Для цього ви додаєте іншу змінну, яка міститиме фон спрайт.

```
var Game = {
canvas : undefined, canvasContext : undefined, backgroundSprite : undefined,
balloonSprite : undefined,
balloonPosition : { x : 0, y : 50 }
};
```

Крім того, в методі draw є два виклики методу drawImage, а не один:

```
Game.draw = function () { Game.drawImage(Game.backgroundSprite, { x : 0, y
: 0 }); Game.drawImage(Game.balloonSprite, Game.balloonPosition);
};
```

Порядок називання цих методів дуже важливий! Оскільки ви хочете, щоб куля з'явилася на тлі фону, *спочатку* потрібно намалювати фон, а *потім* намалювати кулю.Якщо ви зробили це навпаки, тло буде намальовано над кулею, і ви більше не побачите його (спробуйте самі). Малюнок 2.2 показує вихід програми.

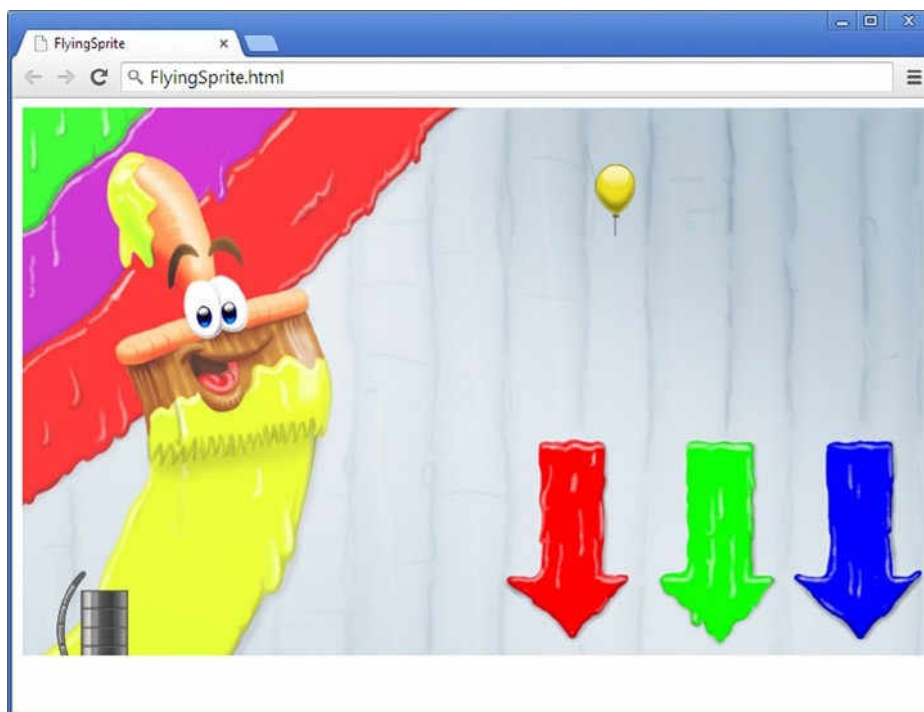


Рисунок 2.2 - Screen shot: *Вихід програми FlyingSprite*

Кожного разу як ви хочете малювати спрайт на в екран ви додаєте drawImage метод в в draw метод.

Game.draw = function () { Game.drawImage(Game.backgroundSprite, { x : 0, y : 0 }); Game.drawImage(Game.balloonSprite, { x : 0, y : 0 }); Game.drawImage(Game.balloonSprite, { x : 100, y : 0 }); Game.drawImage(Game.balloonSprite, { x : 200, y : 0 }); Game.drawImage(Game.balloonSprite, { x : 0, y : 300 }); Game.drawImage(Game.balloonSprite, { x : 200, y : 300 });
};

Знову зверніть увагу на порядок, в якому ви малюєте спрайтів.

Ви також можете одночасно малювати кілька рухомих спрайтів. Для кожної кулі ви можете визначити свою власну змінну положення, яка оновлюється в методі update:

Game.update = function () { var d = new Date();
Game.balloonPosition1.x = d.getTime() % Game.canvas.width; Game.balloonPosition2.x = (d.getTime() + 100) % Game.canvas.width; Game.balloonPosition3.x = (d.getTime() + 200) % Game.canvas.width;
};

А в методі дро, ви використовуєте ці позиції , щоб зробити переміщення і статичні повітряні кулі в той же час:

Game.draw = function () {

```
Game.drawImage(Game.backgroundSprite, Game.balloonPosition1);
Game.drawImage(Game.balloonSprite, Game.balloonPosition2);
Game.drawImage(Game.balloonSprite, Game.balloonPosition3);
Game.drawImage(Game.balloonSprite, { x : 200, y : 0 });
Game.drawImage(Game.balloonSprite, { x : 0, y : 300 });
Game.drawImage(Game.balloonSprite, { x : 200, y : 300 });
};
```

Пограйте з прикладом. Подумайте про різні способи малювати рухомі кулі на екрані. Спробуйте кілька різних значень позиції.Чи можете ви дозволити деяким кулькам рухатися швидше або повільніше, ніж інші?

Музика та звуки

Інший тип звичайного ігрового активу *звучить* .Більшість ігор містять звукові ефекти та фонову музику. Вони важливі з різних причин. Звукові ефекти дають важливі сигнали, які вказують користувачеві, що щось сталося. Наприклад, відтворення звуку кліку, коли користувач натискає кнопку, надає користувачеві зворотний зв'язок з тим, що ця кнопка дійсно була натиснута. Слухові кроки вказують на те, що вороги можуть бути поблизу, навіть якщо гравець може їх ще не бачити.І, чуючи звучання дзвінка на відстані, можна побачити, що щось станеться. Стара гра Myst була класичною у цьому відношенні, тому що багато київ про те, як прогрес було передано гравцеві через звуки

Атмосферні звукові ефекти, такі як капання води, вітер у деревах, а також звучання автомобілів у далечині, покращують досвід і дають відчуття присутності в ігровому світі. Вони роблять середовище живим навіть тоді, коли на екрані нічого не відбувається.

В JavaScript дуже легко грати фонову музику або звукові ефекти.Щоб використовувати звук, вам спочатку потрібно звук файл це ви може грати В програмі гратиме в файл snd_music.mp3 ,

який служить фоновою музикою. Подібно до зберігання та використання спрайтів, ви додаєте змінну до ігор

об'єкт, в якому зберігаються музичні дані.Отже, game об'єкт оголошується та ініціалізується таким чином:

```
var Game = {
canvas : undefined, canvasContext : undefined, backgroundSprite : undefined, balloonSprite : undefined,
balloonPosition : { x : 0, y : 50 }, backgroundMusic : undefined
};
```

Щоб завантажити звуковий ефект або фонову музику, потрібно додати кілька інструкцій до методу start .JavaScript забезпечує тип, який можна використовувати як чернетку для створення об'єкта, що представляє звук. Цей

тип називається Audio .Ви можете створити об'єкт такого типу і починати завантаження звуку як наступним чином:

Game.backgroundMusic = new Audio(); Game.backgroundMusic.src = "snd_music.mp3";

Як видно, це працює майже так само, як і завантаження спрайту. Тепер ви можете викликати методи, які визначаються як частина цього об'єкта, і ви можете встановити змінні учасника об'єкта. Наприклад, наступна інструкція повідомляє браузеру про початок відтворення аудіо, що зберігається в Gae. backgroundMusic змінна:

Game.backgroundMusic.play ();

Ви хочете зменшити гучність фонової музики, щоб пізніше ви могли грати (голосніше) звукові ефекти .Встановлення гучності виконується наступним чином інструкція:

Game.backgroundMusic.volume = 0.4;

Volume , як правило , має значення між 0 і 1, де 0 означає відсутність звуку і 1 відтворює звук в повному обсязі.

Технічно немає різниці між фоновою музикою та звуковими ефектами.Зазвичай фонова музика відтворюється на нижчому рівні; і багато ігор петлять фонову музику так що коли в пісня закінчується вона запускається заново.

Практичне заняття 3
Об'єктно-орієнтований підхід до створення ігор мовою JavaScript

Анотація. Практичне заняття орієнтовано на отримання студентами практичних навичок створення та управління об'єктами мовою JavaScript.

Мета практичного заняття:

- Надати студентам поняття концепції класу.
- Навчити студентів додавати елемент випадковості в ігрові веб-додатки.

У разі успішного проходження практичного заняття студент буде знати основні прийоми роботи з класами та додавати елемент випадковості в ігрові веб-додатки мовою програмування JavaScript.

Вступ

У статті розглядаються принципи об'єктно-орієнтованого програмування (ООП) на JavaScript, а також описуються прототипна і класична моделі успадкування. Наведені в статті приклади ілюструють зустрічаються при створенні ігор типові шаблони, які можуть значно виграти від структурованості і зручності супроводу ООП-підходу. Кінцева мета полягає в тому, щоб кожен фрагмент програмного коду був би доступний для прочитання людиною, висловлював би ідею розробника і служив би спільної мети, в результаті чого вся сукупність інструкцій і алгоритмів перетворилася б в гармонійне твір мистецтва.

Огляд ООП в JavaScript

Часто використовувані скорочення

- DOM: Document Object Model (об'єктна модель документа)
- DRY: Do not Repeat Yourself (не повторювати - принцип розробки програмного забезпечення)
- OOP: Object-oriented programming (об'єктно-орієнтоване програмування)

Мета об'єктно-орієнтованого програмування (ООП) полягає в тому, щоб забезпечити функціонування таких концепцій, як абстракція, модульність, інкапсуляція, поліморфізм і успадкування. ООП дозволяє абстрагувати концепцію програмного коду від його безпосередньої розробки і тим самим забезпечити йому елегантність, можливість багаторазового використання і легкість для читання - ціною збільшення кількості файлів і кількості рядків коду, а також (в разі поганого управління) зниження продуктивності.

Традиційно розробники ігор уникали чистих ООП-підходів, що дозволяло їм вичавлювати максимально можливу продуктивність з кожного циклу центрального процесора. Автори багатьох навчальних посібників з метою швидкого створення демонстраційних версій JavaScript-ігор використовують не відповідають принципам ООП підходи - замість того, щоб надати міцний фундамент. Проблеми розробників ігор на JavaScript відрізняються від проблем розробників для інших середовищ: управління пам'яттю здійснюється не в ручному режимі, а JavaScript-файли виконуються в глобальному контексті, що породжує кошмари супроводу у вигляді заплутаного програмного коду, конфліктів просторів імен і лабіринтоподібного конструкцій if / else. Щоб реалізувати максимум можливого при розробці JavaScript-ігри, застосовуйте найкращі методики ООП.

прототипна спадкування

На відміну від мов, в яких використовується класична спадкування (від слова "клас"), в JavaScript вбудована конструкція класу відсутній. У світі JavaScript "об'єктами першого класу" (first-class object) є функції; як і у всіх визначених користувачем об'єктів, у них є прототипи. Виклик будь-якої функції з ключовим словом new фактично створює копію об'єкта-прототипу цієї функції і використовує цей об'єкт в якості контексту для ключового слова this всередині цієї функції. Відповідний приклад показаний в лістингу 1 .

Лістинг 1. Конструювання об'єкта за допомогою прототипів

```
// constructor function
function MyExample () {
  // property of an instance when used with the 'new' keyword
  this.isTrue = true;
};

MyExample.prototype.getTrue = function () {
  return this.isTrue;
}

MyExample ();
// here, MyExample was called in the global context,
// so the window object now has an isTrue property-this is NOT a good practice

MyExample.getTrue;
// this is undefined-the getTrue method is a part of the MyExample prototype,
// not the function itself

var example = new MyExample ();
// example is now an object whose prototype is MyExample.prototype

example.getTrue; // evaluates to a function
example.getTrue (); // evaluates to true because isTrue is a property of the
                // example instance
```

Згідно зі встановленими нормами, функція, що представляє клас, повинна починатися з великої літери, щоб тим самим показати, що вона призначена для використання в якості конструктора. Ім'я функції має відображати структуру даних, яку вона створює.

Створення екземплярів класів здійснюється за допомогою поєднання ключового слова new і об'єктів-прототипів. Об'єкти-прототипи можуть мати як методи, так і властивості (див. Лістинг 2).

Лістинг 2. Просте успадкування від прототипу

```
// Base class
function Character () {};

Character.prototype.health = 100;

Character.prototype.getHealth = function () {
  return this.health;
}

// Inherited classes
```

```
function Player () {
  this.health = 200;
}

Player.prototype = new Character;

function Monster () {}

Monster.prototype = new Character;

var player1 = new Player ();

var monster1 = new Monster ();

player1.getHealth (); // 200 assigned in constructor

monster1.getHealth (); // 100 inherited from the prototype object
```

Для присвоєння батьківського класу дочірньому класу потрібно здійснити виклик з ключовим словом newі привласнити результат властивості prototypeдочірнього класу (див. Лістинг 3). Внаслідок цього рекомендується в максимально можливій мірі робити конструктори компактними і вільними від побічних ефектів - до тих пір, поки ви не збираєтеся в своїх визначеннях класів передавати значення за замовчуванням.

Якщо ви вже намагалися визначати класи і успадкування на мові JavaScript, то напевно усвідомили його основна відмінність від класичних ООП-мов: в ньому відсутні властивості superі parent, які дозволяли б отримати доступ до методів батьківського об'єкта в разі їх перевизначення. Для цієї ситуації існує просте рішення, проте воно порушує принципи DRY, що, ймовірно, і є основною причиною наявності такої великої кількості бібліотек, які намагаються імітувати класичне успадкування.

Лістинг 3. Виклик методів батька з дочірніх класів

```
function ParentClass () {
  this.color = 'red';
  this.shape = 'square';
}

function ChildClass () {
  ParentClass.call (this); // use 'call' or 'apply' and pass in the child
                          // class's context
  this.shape = 'circle';
}
```

```
ChildClass.prototype = new ParentClass (); // ChildClass inherits from ParentClass

ChildClass.prototype.getColor = function () {
  return this.color; // returns "red" from the inherited property
};
```

У лістингу 3 значення атрибутів colori shape знаходяться не в прототипі - вони присвоюються в функції конструктора ParentClass. Властивості shape в нових примірниках ChildClass значення присвоюється два рази - один раз в конструкторі ParentClassйому присвоюється значення square і один раз в конструкторі ChildClassйому присвоюється значення circle. Переміщення в прототип такої логіки, як ці присвоєння, послабить вищеописані побічні ефекти і полегшить супровід програмного коду.

У моделях прототипного успадкування для виконання функції з різним контекстом можна використовувати JavaScript-методи calli apply Цей підхід добре працює в якості заміни використання властивостей super і parent, наявних в інших мовах, однак він породжує іншу проблему. Якщо вам необхідно здійснити рефакторинг класу, наприклад, змінити його ім'я, його батьків або ім'я його батька, тепер у вас є маркер ParentClassу відповідному текстовому файлі в набагато більшій кількості місць. Ця проблема посилюється в міру того, як ваші класи стають все більш складними. Привабливіше рішення полягає в тому, щоб ваші класи розширювали якийсь базовий клас, що дозволило б вашому коду повторювати себе меншу кількість разів, зазвичай у формі відтворення класичного наслідування.

класичне спадкування

Хоча прототипна спадкування повністю підходить для використання в ООП, воно не задовольняє декільком критеріям хорошого стилю програмування. Зокрема, воно має такі недоліки.

- Воно не відповідає принципам DRY. Імена класів і прототипи повторюються буквально всюди, що ускладнює читання і рефакторинг.
- Виклик конструкторів здійснюється під час визначення прототипу. Ви не можете використовувати певну логіку в конструкторах після того, як починаєте розбиття на підкласи.
- Відсутня реальна підтримка сильної інкапсуляції.
- Відсутня реальна підтримка статичних членів класу.

Багато JavaScript-бібліотеки намагаються дотримуватися більш класичний синтаксис ООП, щоб подолати вищезазначені проблеми. Одна з таких простих у використанні бібліотек - Base.js, творцем якої є Дін Едвард (Dean Edward) (див. Розділ Ресурси). Ця бібліотека надає наступні корисні можливості.

- Все прототипирование здійснюється за допомогою т.зв. "Об'єктів-домішок" (object mix-ins); (Класи і підкласи можуть бути визначені в одному затвердження).
- Для надання безпечного місця для логіки, яка підлягає виконанню при

створенні нових екземплярів класу, використовується спеціальна функція конструктора.

- Тим самим реалізується підтримка статичних членів класу.
- Внесок цієї бібліотеки в сильну инкапсуляцию закінчується на мінімізації визначення класу до одного твердження (інкапсуляція думки замість інкапсуляції коду).

Інші бібліотеки можуть пропонувати більш сувору підтримку публічних і приватних методів і властивостей (інкапсуляція), однак бібліотека Base.js надає короткий синтаксис, простий у використанні і легкий для запам'ятовування.

Лістинг 4 являє собою короткий вступ в Base.js і в класичне успадкування. У цьому прикладі характеристики абстрактного класу Enemyрозширюються за допомогою більш певного специфічного класу RobotEnemy.

Лістинг 4. Короткий вступ в Base.js і в класичне спадкування

```
// create an abstract, basic class for all enemies
// the object used in the .extend () method is the prototype
var Enemy = Base.extend ({
   health: 0,
   damage: 0,
   isEnemy: true,

   constructor: function () {
      // this is called every time you use "new"
   },

   attack: function (player) {
      player.hit (this.damage); // "this" is your enemy!
   }
});

// create a robot class that uses Enemy as its parent
//
var RobotEnemy = Enemy.extend ({
   health: 100,
   damage: 10,

   // because a constructor is not listed here,
   // Base.js automatically uses the Enemy constructor for us

   attack: function (player) {
      // you can call methods from the parent class using this.base
      // by not having to refer to the parent class
      // or use call / apply, refactoring is easier
```

```
        // in this example, the player will be hit
        this.base (player);

        // even though you used the parent class's "attack"
        // method, you can still have logic specific to your robot class
        this.health + = 10;
    }
});
```

ООП-шаблони при проектуванні ігор

Базовий механізм гри незмінно спирається на наступні дві функції: updatei render. У свою чергу, метод renderзазвичай використовує метод setIntervalабо polyfill-скрипт для API-інтерфейсу requestAnimationFrame, напр., Створений Полом Айріш (Paul Irish) (див. Розділ Ресурси). Перевага використання інтерфейсу requestAnimationFrameполягає в тому, що його виклик здійснюється не частіше, ніж це необхідно. Він буде виконуватися лише з частотою оновлення монітора користувача (як правило, 60 разів на секунду для настільних систем), а в більшості браузерів він взагалі не буде виконуватися, якщо вкладка, в якій знаходиться гра, не є активною. переваги:

- Скорочення обсягу виконуваної клієнтським комп'ютером роботи в той час, коли користувач не дивиться на гру.
- Максимальне збільшення строку служби акумулятора на мобільних пристроях.
- Ефективний механізм припинення гри, якщо її цикл оновлення (update) пов'язаний з циклом рендеринга (render).

За вищезгаданих причин застосування requestAnimationFrameвважається дружнім до клієнта і кращим у порівнянні з setInterval.

Зв'язування циклу updateз циклом renderпороджує іншу проблему: підтримка однаковою частоти для ігрових дій і для анімацій незалежно від того, чи здійснюється цикл рендеринга з частотою 15 або 60 кадрів в секунду. Трюк для подолання цієї проблеми полягає в тому, щоб всередині гри ввести одиницю часу під назвою такт (*tick*) І передавати кількість часу, що пройшов з моменту останнього дзвінка до оновлення. Потім ця кількість часу може бути перетворено в кількість тактів, після чого моделі, фізичні механізми і інша залежна від часу ігрова логіка будуть налаштовані відповідним чином. Наприклад, отруєний гравець може отримувати по 10 балів шкоди на кожному такті на протязі 10 тактів. Якщо цикл рендеринга працює швидко, то до моменту поновлення певного виклику збиток може взагалі не настати. Однак якщо на останньому циклі рендеринга буде активована прибирання сміття, що викликає пропуск півтора тактів, то ігрова логіка може, навпаки, привласнити гравцеві збиток величиною 15 балів.

Інший підхід полягає в повному відділенні синхронізації оновлень моделі від циклу рендеринга зображення. В іграх, що мають велику кількість анімацій або об'єктів, або з тих чи інших причин витрачають велику кількість ресурсів на

малювання, зв'язування циклу оновлення з циклом рендеринга призведе до загального сповільнення гри. У цьому випадку метод updateможе працювати на інтервалі установки (з використанням методу setInterval) незалежно від того, коли і як часто запускається обробникrequestAnimationFrame. Велика частина часу, що витрачається в цих циклах, фактично витрачається на кроці рендеринга, тому гра продовжить функціонування із заданою швидкістю, навіть якщо малювання на екрані здійснюється з частотою всього лише 25 кадрів / с. В обох випадках розробнику як і раніше необхідно обчислювати різницю в часі між циклами оновлення; якщо оновлення проводиться 60 раз в секунду, то у функції поновлення є до завершення не більше 16 мс. Якщо ця функція буде виконуватися довше (або якщо в браузері буде запущена збірка сміття), то гра буде як і раніше сповільнюватися. Відповідний приклад показаний в лістингу 5 .

Лістинг 5. Базовий клас додатки з циклами renderі updateloops

```
// requestAnim shim layer by Paul Irish
   window.requestAnimFrame = (function () {
     return window.requestAnimationFrame ||
           window.webkitRequestAnimationFrame ||
           window.mozRequestAnimationFrame ||
           window.oRequestAnimationFrame ||
           window.msRequestAnimationFrame ||
           function (/ * function * / callback, / * DOMElement * / element) {
             window.setTimeout (callback, 1000/60);
           };
   }) ();

var Engine = Base.extend ({
   stateMachine: null, // state machine that handles state transitions
   viewStack: null, // array collection of view layers,
                    // perhaps including sub-view classes
   entities: null, // array collection of active entities within the system
                    // characters,
   constructor: function () {
      this.viewStack = []; // do not forget that arrays should not be prototype
                           // properties as they're copied by reference
      this.entities = [];

      // set up your state machine here, along with the current state
      // this will be expanded upon in the next section

      // start rendering your views
      this.render ();
     // start updating any entities that may exist
     setInterval (this.update.bind (this), Engine.UPDATE_INTERVAL);
```

```
        },

        render: function () {
            requestAnimFrame (this.render.bind (this));
            for (var i = 0, len = this.viewStack.length; i <len; i ++) {
                // delegate rendering logic to each view layer
                (This.viewStack [i]). Render ();
            }
        },

        update: function () {
            for (var i = 0, len = this.entities.length; i <len; i ++) {
                // delegate update logic to each entity
                (This.entities [i]). Update ();
            }
        }
    },

    // Syntax for Class "Static" properties in Base.js. Pass in as an optional
    // second argument to.extend ()
    {
        UPDATE_INTERVAL: 1000/16
    });
```

Якщо ви не знайомі з контекстом thisв JavaScript, то зверніть увагу, що .bind(this)використовується двічі - один раз в анонімній функції всередині виклику setIntervali один раз в this.render.bind()середині виклику requestAnimFrame. В даному випадкуsetIntervali requestAnimFrame- це функції, а не методи; вони належать глобальному об'єкту window, а не якогось певного класу або сутності. В результаті для того, щоб thisусередині методів render і update ігрового механізму посилався на наш екземпляр класуEngine, виклик .bind(object)змушує thisвсередині функції діяти не так, як зазвичай. При необхідності підтримки браузера Internet Explorer версії 8 або нижче необхідно додати відповідний polyfill-скрипт для реалізації зв'язування.

<u>На початок</u>

Шаблон State Machine

Шаблон State Machine (state machine - машина з кінцевим числом станів) широко застосовується, хоча це не завжди усвідомлюється. Даний шаблон являє собою розширення принципів ООП (абстрагування концепції програмного коду від його виконання). Наприклад, гра може мати такі стани:

- Попереднє завантаження
- початковий екран
- активна гра
- Меню опцій

• Кінець гри (перемога, програш, продовження)

Ні в одному з цих станів не повинно бути виконуваного коду, що залежить від інших станів. Наприклад, програмний код попереднього завантаження нічого не повинен знати про те, коли слід відкривати меню опцій. Імперативне (процедурне) програмування могло б запропонувати монолітні умовні оператори if або switch, щоб належним чином впорядкувати логіку додатка, однак такі оператори не в змозі представити концепцію програмного коду, що ускладнює подальший супровід. Введення додаткових станів, таких як внутріігрового меню, переходи між рівнями, нові функції і т.д., в ще більшому ступені ускладнює супровід умовних операторів.

Замість цього розглянемо приклад в лістингу 6 .

Лістинг 6. Спрощена машина з кінцевим числом станів

```
// State Machine
var StateMachine = Base.extend ({
    states: null, // this will be an array, but avoid arrays on prototypes.
                  // as they're shared across all instances!
    currentState: null, // may or may not be set in constructor
    constructor: function (options) {
        options = options || {}; // optionally include states or contextual awareness

        this.currentState = null;
        this.states = {};

        if (options.states) {
            this.states = options.states;
        }

        if (options.currentState) {
            this.transition (options.currentState);
        }
    },

    addState: function (name, stateInstance) {
        this.states [name] = stateInstance;
    },

    // This is the most important function-it allows programmatically driven
    // changes in state, such as calling myStateMachine.transition ( "gameOver")
    transition: function (nextState) {
        if (this.currentState) {
            // leave the current state-transition out, unload assets, views, so on
            this.currentState.onLeave ();
        }
```

```
            // change the reference to the desired state
            this.currentState = this.states [nextState];
            // enter the new state, swap in views,
            // setup event handlers, animated transitions
            this.currentState.onEnter ();
        }
    });

    // Abstract single state
    var State = Base.extend ({
        name: '', // unique identifier used for transitions
        context: null, // state identity context- determining state transition logic

        constructor: function (context) {
            this.context = context;
        },

        onEnter: function () {
            // abstract

            // use for transition effects
        },

        onLeave: function () {
            // abstract

            // use for transition effects and / or
            // memory management- call a destructor method to clean up object
            // references that the garbage collector might not think are ready,
            // such as cyclical references between objects and arrays that
            // contain the objects
        }
    });
```

Можливо, у вас немає необхідності для свого додатку створювати спеціальний підклас класу state machine, однак ви напевно захочете створити підкласи Stateдля кожного з станів свого застосування. Поділ логіки переходів на різні об'єкти дозволяє:

- Використовувати конструктори як можливості для негайного початку попереднього завантаження активів.
- Додавати до гри нові стану (наприклад, екран продовження, який з'являється перед екраном закінчення гри) без необхідності з'ясовувати, на які глобальні змінні впливають ті чи інші умовні вирази в тих чи інших монолітних структурах if / else або switch.

• Динамічно визначати логіку переходів в разі створення станів на основі даних, що завантажуються з сервера.

Основний клас додатків не повинен турбуватися про логіку всередині станів, а ці стани не повинні надто турбуватися про основне класі програми. Наприклад, стан попереднього завантаження може відповідати за формування уявлення на основі активів, вбудованих в розмітку сторінки, і за формування черг мінімально необхідних ігрових активів (відеокліпи, зображення та звуки) в диспетчері singleton-активів. Хоча цей стан задіє клас попереднього завантаження уявлення, вона не зобов'язана піклуватися про поведінку цього подання. В даному випадку ідея (об'єкт, що представляється станом) полягає в обмеженні відповідальності,-необхідно лише визначити, як має поводитися додаток в змозі попереднього завантаження даних.

Слід мати на увазі, що шаблон state machine не обмежений станами ігровий логіки. Окремі уявлення також виграють від виключення логіки стану з логіки уявлення, особливо при управлінні субпредставленіямі або при об'єднанні з шаблоном Chain of Responsibility з метою обробки подій, що виникають при взаємодії з користувачем.

Шаблон Chain of Responsibility: емуляція спливаючих подій на елементі canvas

HTML5-елемент canvasможна розглядати як елемент зображення, який дозволяє маніпулювати окремими пікселями. Наприклад, якщо в якій-небудь області екрану намальована трава, що у ньому награбоване видобуток і стоїть на цьому персонаж, то canvas не зможе здогадатися, з якого їх цих предметів натиснув користувач. Якщо ви намалювали меню, то canvas не знає, що певна його область являє собою кнопку, і що єдиний DOM-елемент, з яким може бути пов'язана подія, і є сам canvas. Щоб в гру дійсно можна було б грати, ігровий механізм повинен розуміти, що може відбуватися при натисканні користувача на canvas.

Шаблон проектування Chain of Responsibility покликаний відокремити відправника події (DOM-елемент) від одержувача (розробляється код), щоб відповідальність за обробку події міг би взяти на себе більш ніж один об'єкт (уявлення і моделі). У класичних реалізаціях, таких як веб-сторінки, уявлення і моделі можуть мати обробляє інтерфейс і з його допомогою делегувати всі породжувані разів клацнувши мишкою події графу сцени, який після цього знаходить відповідні "речі", на які натиснув користувач, і надає кожній з них шанс на переривання. Простіший підхід полягає в тому, що на самому елементі canvas розмістити ланцюжок обробників, що визначаються в процесі виконання (див. Лістинг 7).

Лістинг 7. Обробка спливаючих подій з використанням шаблону Chain of Responsibility

```
var ChainOfResponsibility  = Base.extend ({
        context: null, // relevant context- view, application state, so on
        handlers: null, // array of responsibility handlers
```

```
        canPropagate: true, // whether or not

        constructor: function (context, arrHandlers) {
           this.context = context;
           if (arrHandlers) {
              this.handlers = arrHandlers;
           } Else {
              this.handlers = [];
           }
        },

        execute: function (data) {
           for (var i = 0, len = this.handlers.length; i <len; i ++) {
              if (this.canPropagate) {
                 // give a handler a chance to claim responsibility
                 (This.handlers [i]). Execute (this, data);
              } Else {
                 // an event has claimed responsibility, no need to continue
                 break;
              }
           }
           // reset state after event has been handled
           this.canPropagate = true;
        },

        // this is the method a handler can call to claim responsibility
        // and prevent other handlers from acting on the event
    stopPropagation: function () {
           this.canPropagate = false;
        },

        addHandler:  function (handler) {
           this.handlers.push (handler);
        }
    });

    var ResponsibilityHandler  = Base.extend ({
       execute: function (chain, data) {

        // use chain to call chain.stopPropegation () if this handler claims
        // responsibility,  or to get access to the chain's context member property
        // if this event handler does not need to claim responsibility, simply
        // return; and the next handler will execute
       }
```

```
});
```

Клас ChainOfResponsibilityбуде прекрасно працювати і без поділу на підкласи, оскільки вся специфічна логіка додатка буде міститися в підкласах ResponsibilityHandler. Єдина відмінність між реалізаціями полягає в передачі всередину відповідного контексту, наприклад, відображення, яку представляють цим контекстом. Розглянемо меню опцій, при відкритті якого призупинена гра як і раніше демонструється на екрані (див. Лістинг 8). Якщо користувач натискає на будь-яку з кнопок цього меню, персонажі на задньому плані не повинні реагувати на це натискання.

Лістинг 8. Обробник закриття меню опцій

```
var OptionsMenuCloseHandler  = ResponsibilityHandler.extend  ({
   execute: function (chain, eventData) {
      if (chain.context.isPointInBackground (eventData)) {
         // the user clicked the transparent background of our menu
         chain.context.close (); // delegate changing state to the view
          chain.stopPropegation (); // the view has closed, the event has been
handled
      }
   }
});

// OptionMenuState
// Our main view class has its own states, each of which handles
// which chains of responsibility are active at any time as well
// as visual transitions

// Class definition ...
constructor: function () {
   // ...
   this.chain = new ChainOfResponsibility  (
      this.optionsMenuView, // the chain's context for handling responsibility
      [
         new OptionsMenuCloseHandler  (), // concrete implementation of
                                           // a ResponsibilityHandler
         // ... other responsibility handlers ...
      ]
   );
}

// ...
onEnter: function () {
   // change the view's chain of responsibility
   // guarantees only the relevant code can execute
```

```
    // other states will have different chains to handle clicks on the same view
    this.context.setClickHandlerChain (this.chain);
  }
  // ...
```

У лістингу 8 клас viewмає посилання на набір станів, а кожне стан визначає, які об'єкти будуть відповідати за конкретну обробку події натискання. В результаті логіка подання є обмеженою - в її розпорядженні є тільки те, що представляють її сутності, тому воно відображає тільки меню опцій. Якщо в результаті поновлення в грі з'являються додаткові кнопки, нові ефекти або переходи до нових уявлень, то для кожної нової "особливості" з'являється окремий об'єкт, здатний обробляти цю особливість без будь-якої потреби змінювати, порушувати чи переписувати існуючу логіку. Розумне поєднання ланцюжків для таких подій, як переміщення і натискання миші, дозволяє здійснювати обробку буквально всім - від меню і символів до перетягування екранів з "майном"

На початок

висновок

За своєю суттю концепції шаблонів проектування і об'єктно-орієнтованого програмування (ООП) є нейтральними; їх непродумане застосування здатне породжувати нові проблеми, замість того щоб вирішувати існуючі. У даній статті були розглянуті принципи ООП при використанні JavaScript, а також описані прототипна і класична моделі успадкування. Ви дізналися про широко застосовуваних при створенні ігор шаблонах, які здатні мати істотну вигоду з структурованості і зручності супроводу ООП-дизайну (базовий ігровий цикл, машина з кінцевим числом станів і спливаючі події). Ця стаття торкнулася лише поверхні типових рішень для типових проблем. Після невеликої практики ви станете фахівцем з написання виразного коду - і, відповідно, будете витрачати менше часу на написання і більше часу на творчість.

Практичне заняття 4
Контрольований доступ до об'єктів

Анотація. Практичне заняття орієнтовано на оволодіння студентами навичками створення модифікаторів доступу до властивостей об'єктів.

Мета практичного заняття:

- Навчити студентів прийомам створення модифікаторів доступу до властивостей об'єктів.
- Навчити студентів підтримувати колізії між ігровими об'єктами.

У разі успішного виконання лабораторної роботи студент буде вміти створювати модифікатори доступу до властивостей об'єктів мовою JavaScript.

Організація гри Об'єкти

Ви вже бачили в попередніх розділах, як можна використовувати класи для групових змінних, які належать разом. У цьому розділі розглядається схожість між різними типами ігрових об'єктів і як ви можете виразити ці подібності в JavaScript.

Подібності між Гра Об'єкти

Якщо ви дивитесь на різні ігрові об'єкти у грі Painter, ви можете побачити, що у них багато спільного. Наприклад, м'яч, гармата та банки для фарбування використовують три спрайти, які представляють кожен з трьох різних кольорів. Крім того, більшість об'єктів у грі мають позицію і швидкість. Крім того, для всіх ігрових об'єктів потрібен спосіб їх малювати, деякі з об'єктів гри мають метод обробки введення, деякі з них мають метод update тощо.Тепер це не проблема, що ці класи мають схожість.Браузер чи гравці гри не скаржаться на це. Тим НЕ менше, це дуже шкода , що ви повинні скопіювати код весь час.Як приклад, як ball і в PaintCan клас мати width і height властивості:

```
Object.defineProperty(Ball.prototype, "width",
{
get: function () {
return this.currentColor.width;
}
});
Object.defineProperty(Ball.prototype, "height",
{
get: function () {
return this.currentColor.height;
}
});
```

Код точно такий же, але ви повинні скопіювати його для обох класів.І кожного разу, коли ви хочете додати інший вид об'єкта гри, вам, можливо, доведеться скопіювати ці властивості знову. У цьому випадку ці властивості, на щастя, не настільки складні, але в додатку ви копіюєте багато іншого. Наприклад, більшість класів ігрових об'єктів в іграх Painter визначають наступний член змінні:

this.currentColor = *some sprite*; this.velocity = Vector2.zero; this.position = Vector2.zero; this.origin = Vector2.zero; this.rotation = 0;

Методи draw різних ігрових об'єктів теж схожі.Наприклад, ось draw методи Ball та PaintCan класи:

```
Ball.prototype.draw = function () { if (!this.shooting)
return;
Canvas2D.drawImage(this.currentColor, this.position, this.rotation, 1,
this.origin);
};
PaintCan.prototype.draw = function () {
Canvas2D.drawImage(this.currentColor, this.position, this.rotation, 1,
this.origin);
};
```

Знову ж таки, код дуже подібний у різних класах, і ви копіюєте його щоразу, коли ви створюєте новий вид об'єкта гри. Загалом краще уникати копіювання багатьох кодів. Чому так? Оскільки

якщо в якийсь момент ви розумієте, що в цій частині коду є помилка, ви повинні виправити це скрізь, до якого ви його скопіювали. У невеликій грі, як Painter, це не велика проблема.Але коли ви розробляєте комерційну гру із сотнями різних класів ігрових об'єктів, це стає серйозною проблемою підтримки. Крім того, ви не завжди знаєте, наскільки далеко буде маленька гра. Якщо ви не будете обережні, можна скопіювати багато коду (і пов'язані з ним помилки). Як гра дозріває, це гарна ідея , щоб стежити за де оптимізувати код, навіть якщо це означає деяку додаткову роботу , щоб знайти ці дуплікації і закріпити їх.Для цієї конкретної ситуації, ви треба

щоб подумати про те, як різні види ігрових об'єктів подібні, і чи можете ви об'єднати ці подібності разом, так само як і згрупував змінні-члени в попередніх розділах.

Концептуально кажучи, це легко сказати , що схоже між кульками, банки з фарбою, і гарматами: вони всі *ігрові об'єкти.В принципі, вони можуть бути намальовані на певній посаді; вони всі мають швидкість (навіть гармата, але її швидкість дорівнює нулю); і всі вони мають колір, який є червоним, зеленим або* синій

Крім того, більшість з них вдаються до введення деяких видів і оновлюються.

Спадкування

Використовуючи прототипи в JavaScript, можна об'єднати ці подібності у певний загальний клас, а потім визначити інші класи, які є *спеціальною версією* цього родового класу.У об'єктно-орієнтованому жаргоні це називається *спадщиною* , і це дуже потужна функція мови.У JavaScript спадщина стає можливим за прототипом механізму. Розглянемо наступний приклад:

```
function Vehicle() { this.numberOfWheels = 4; this.brand = "";
```

```
}
Vehicle.prototype.what = function() {
return "nrOfWheels = " + this.numberOfWheels + ", brand = " + this.brand;
};
```

Тут ви маєте дуже простий приклад класу для представлення транспортних засобів (ви можете уявити, що це може бути корисним для гри, що моделює трафік). Щоб тримати все простіше, автомобіль визначається кількома колесами та брендом.Клас автомобіля також має метод, який називається what що повертає а

опис транспортного засобу. Це може бути корисним, якщо ви хочете створити веб-сайт, який представив список транспортних засобів у таблиці. Ви можете використовувати цей клас таким чином:

```
var v = new Vehicle(); v.brand = "volkswagen";
console.log(v.what()); // outputs "nrOfWheels = 4, brand = volkswagen"
```

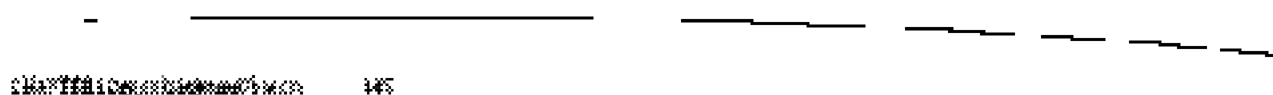
Є різні типи транспорту такі як автомобілі велосипеди мотоцикли і так далі. Для деяких ви би хотіли зберегти додаткову інформацію.Наприклад, для автомобіля, це може бути корисно зберігати чи він є конвертований;

```
function Car(brand) { Vehicle.call(this); this.brand = brand; this.convertible = false;
}
Car.prototype = Object.create(Vehicle.prototype);

var c = new Car("mercedes");
console.log(c.what()); // outputs "nrOfWheels = 4, brand = mercedes"
```

У конструкторі автомобіля є така лінія:

```
Vehicle.call(this);
```



Те, що відбувається тут, полягає в тому, що конструктор транспортного засобу викликається, *використовуючи той самий об'єкт,* який створюється при виклику конструктора автомобіля .По суті, ви розповідаєте перекладачеві, що об'єкт Car (це), який ви наразі маніпулюєте , *фактично є об'єктом* Vehicle .Отже, ви можете побачити два важливих аспекти спадщини:

- Існує зв'язок між об'єктами (автомобіль об'єкт також є транспортним засобом).

■ Клас, який успадковує від іншого класу, копіює його функціональність. (Об'єкти автомобіля мають однакові змінні, властивості та методи учасника, як Vehicle об'єкти).

Тому що автомобіль успадковує від транспортного засобу.

```
function Motorbike(brand) { Vehicle.call(this); this.numberOfWheels = 2;
this.brand = brand; this.cylinders = 4;
}
Motorbike.prototype = Object.create(Vehicle.prototype);
```

Мотоцикл теж є своєрідним автомобілем. Клас мотоциклів успадковує від автомобіля і додає власну змінену особу члену, щоб вказати кількість циліндрів.Малюнок 11-1 ілюструє ієрархію класів.Для більш розширеної версії цієї ієрархії див. Малюнок 11-4 .

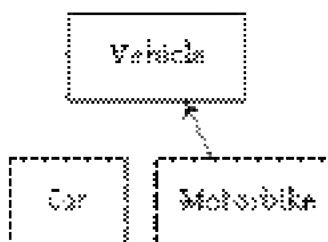


Рисунок 4.1 - *Діаграма успадкування* автомобіля *та його підкласів.*

Ігрові об'єкти та Успадкування

"Подібні" відносини також стосуються ігрових об'єктів у грі Painter. М'яч є своєрідним об'єктом гри, а також банки з фарбою і гармата. Ви можете зробити цю спадщину явним в програмі, визначивши загальний клас з назвою ThreeColorGameObject і отримавши успадкування від фактичного класу ігрового об'єкта з цього родового класу.Потім ви можете поставити все, що визначає, що є об'єктом триколірної гри в цьому класі, і м'яч, гармата та фарба можуть бути спеціальними версіями цього класу.

Давайте розглянемо цей клас ThreeColorGameObject більш детально.Ви вводите в цей клас змінні членів, які зазвичай використовуються різними типами ігрових об'єктів у грі. Ви можете визначити основний скелет цього класу таким чином:

```
function ThreeColorGameObject()
{ this.currentColor = undefined;
this.velocity = Vector2.zero;
this.position = Vector2.zero;
this.origin = Vector2.zero; this.rotation = 0; this.visible = true;
```

```
}
```

Кожен клас, який успадковує від ThreeColorGameObject , матиме швидкість, позицію, походження, обертання тощо.Це добре, оскільки тепер ви лише визначаєте ці змінні-учасники в одному місці, і їх можна використовувати в будь-якому класі, який успадковує від ThreeColorGameObject .

Одна річ, яка все ще відсутня у цьому конструкторі, це спосіб вирішення трьох різних кольорів. У випадку з Painter, кожен тип гри-об'єкта має три різні спрії, кожен з яких представляє інший колір.Коли ви визначаєте клас ThreeColorGameObject , ви ще не знаєте, які спрайти використовувати, оскільки вони будуть залежати від кінцевого типу об'єкта гри (гармати використовують інші спрайти, крім кульок або банок для фарби).Щоб вирішити це, давайте продовжимо конструктор як наступним чином:

```
function ThreeColorGameObject(sprColorRed, sprColorGreen, sprColorBlue)
{ this.colorRed = sprColorRed;
this.colorGreen = sprColorGreen;
this.colorBlue = sprColorBlue;
this.currentColor = this.colorRed;
this.velocity = Vector2.zero;
this.position = Vector2.zero;
this.origin = Vector2.zero;
this.rotation = 0;
this.visible = true;
}
```

Кожного разу, коли ви успадковуєте від цього класу, ви можете визначити, які значення змінних членів

colorRed, colorGreen і colorBlue має бути.

Тепер вам потрібно визначити основні методи обробки циклів. Метод нанесення ігрового об'єкту є простим. Ви, напевно, помітили, що до класу додано змінну-член- видимий .Ви можете використовувати цю змінну-члену, щоб змінити видимість об'єктів гри. У методі draw ви малюєте спрайт на екрані, лише якщо об'єкт гри повинен бути видимим:

```
ThreeColorGameObject.prototype.draw = function ()
{ if (!this.visible)
return;
Canvas2D.drawImage(this.currentColor, this.position, this.rotation, 1,
this.origin);
};
```

Метод оновлення класу містить одну інструкцію, яка оновлює поточну позицію об'єкту гри:

```
ThreeColorGameObject.prototype.update = function (delta)
{ this.position.addTo(this.velocity.multiply(delta));
};
```

Нарешті, ви додаєте декілька зручних властивостей для отримання та налаштування кольору та отримання розмірів об'єкта. Наприклад, це властивість для читання та запису кольору об'єкта:

```
Object.defineProperty(ThreeColorGameObject.prototype, "color",
{
get: function () {
if (this.currentColor === this.colorRed) return Color.red;
elseif (this.currentColor === this.colorGreen) return Color.green;
else
return Color.blue;
},
set: function (value) {
if (value === Color.red) this.currentColor = this.colorRed;
else if (value === Color.green) this.currentColor = this.colorGreen;
else if (value === Color.blue) this.currentColor = this.colorBlue;
}
});
```

Як ви можете бачити, ви використовуєте кольорові змінні члена спірити тут. Будь-який клас, який успадковує від ThreeColorGameObject, також має цю властивість.Це заощаджує багато копіювання коду! Для повного класу ThreeColorGameObject перегляньте приклад Painter9, що належить до цього розділу.

Гармата як підклас ThreeColorGameObject

Тепер, коли ви створили дуже базовий клас для кольорових об'єктів гри, ви можете повторно використовувати цю основну поведінку для реальних ігрових об'єктів у вашій грі, *наслідуючи* цей клас.Давайте спочатку подивимося на клас Cannon .Оскільки ви визначили основний клас ThreeColorGameObject , ви можете створити клас Cannon як підклас цього класу таким чином:

```
 function Cannon() {
// to do...
}
Cannon.prototype = Object.create(ThreeColorGameObject.prototype);
```

Ви створюєте об'єкт Cannon.prototype , створивши копію ThreeColorGameObject.prototype об'єкта. Хоча все-таки потрібно написати код у

методі конструктора.

Оскільки Cannon успадковує від ThreeColorGameObject , потрібно викликати конструктор класу ThreeColorGameObject .Цей конструктор очікує три параметри. Оскільки ви створюєте об'єкт Cannon , ви хочете, щоб кольорові спрайти гармати передавали цьому конструктору.На щастя, ви можете пройти разом з цими справами в методі виклику , як показано нижче:

```
ThreeColorGameObject.call(this, sprites.cannon_red, sprites.cannon_green,
sprites.cannon_blue);
```

По-друге, ви встановлюєте позицію та походження гармати, як і в оригінальному класі Cannon :

```
this.position = new Vector2(72, 405); this.origin = new Vector2(34, 34);
```

Решта творів (присвоєння трьох колірних спрацьов та ініціалізація інших змінних членів) виконано для вас у конструкторі ThreeColorGameObject !Зверніть увагу , що це важливо , щоб перший виклик конструктора суперкласу перед установкою змінних - членів в підкласі.В іншому випадку значення позиції та походження, які ви виберете для гармати, будуть скинуті до нуля, коли конструктор ThreeColorGameObject є називається

Тепер, коли була визначена нова версія класу Cannon , ви можете почати додавати властивості і методи до класу, як і раніше.Наприклад, тут знаходиться метод handleInput :

```
Cannon.prototype.handleInput = function (delta) { if (Keyboard.down(Keys.R))
this.currentColor = this.colorRed; else if (Keyboard.down(Keys.G))
this.currentColor = this.colorGreen; else if (Keyboard.down(Keys.B))
this.currentColor = this.colorBlue;
var opposite = Mouse.position.y - this.position.y;
var adjacent = Mouse.position.x - this.position.x;
this.rotation = Math.atan2(opposite, adjacent);
};
```

Як ви можете бачити, ви можете отримати доступ до змінних членів, таких як currentColor і rotation без проблем.Оскільки Cannon успадковує від ThreeColorGameObject , він містить ті самі змінні, властивості та методи учасника.

Переоцінка Методи від в Суперклас

Окрім додавання нових методів та властивостей, ви також можете вибрати метод *заміни* в

Cannon класу.Наприклад, ThreeColorGameObject має наступний метод draw:

```
ThreeColorGameObject.prototype.draw = function () { if (!this.visible)
return;
Canvas2D.drawImage(this.currentColor, this.position, this.rotation, 1,
this.origin);
};
```

Для гармат цей метод не робить саме те, що ви хочете. Ви хочете намалювати колір гармати, але ви також хочете намалювати гармату. Заміна методу дуже проста. Ви просто переосмислюєте метод як частину прототипу Cannon :

```
Cannon.prototype.draw = function () { if (!this.visible)
return;
var colorPosition = this.position.subtract(this.size.divideBy(2));
Canvas2D.drawImage(sprites.cannon_barrel, this.position, this.rotation, 1,
this.origin); Canvas2D.drawImage(this.currentColor, colorPosition);
};
```

У об'єктно-орієнтованому жаргоні, коли ви замінюєте метод, успадкований від суперкласу в підкласі, ви кажете, що ви *перевизначаєте* метод.В цьому випадку, перевизначити метод малювання від ThreeColorGameObject.Cannon був створений, у вас є повна гнучкість, яку пропонує JavaScript для зміни цього об'єкта.

Якщо ви подивіться на файл Cannon.js в прикладі Painter9 , що належить до цієї чолі, ви можете побачити , що визначення класу Cannon набагато менше і легше читати , ніж в попередній версії, оскільки всі загальні члени гри-об'єктних поміщений у клас ThreeColorGameObject .

Організація вашого коду в різних класах та підкласах допомагає зменшити копіювання коду та призводить до загальної чистоти конструкцій. Проте існує застереження: структура вашого класу (який клас наслідує, з якого іншого класу) має бути правильним. Пам'ятайте, що класи слід успадковувати лише від інших класів, якщо між класами існує "своєрідна" взаємозв'язок. Щоб проілюструвати це, припустимо, ви хочете додати індикатор у верхній частині екрана, який показує, який колір зараз знаходиться у м'ячі. Ви можете зробити для цього клас, і нехай він успадковується від класу Cannon, тому що він повинен обробляти введення аналогічним чином:

```
function ColorIndicator() { Cannon.call(this, ...);
// etc.
}
```

Однак це дуже погана ідея. Кольоровий індикатор, безумовно, не якийсь гармат, і розробка ваших класів таким чином робить незрозумілим для інших

розробників, якими класами користуються.

Крім того, індикатор кольору також матиме обертання, що не має сенсу.

Діаграми спадкового класу повинні бути логічними та зрозумілими. Кожного разу, коли ви пишете клас, який успадковує від іншого класу, запитайте себе, чи дійсно цей клас "є певним" класом, який ви успадковуєте. Якщо це не так, то вам доведеться переосмислити ваш дизайн.

Кульковий клас

Ви визначаєте новий клас Ball у моді, дуже схожих на клас Cannon .Як у класі Cannon , ви успадковуєте від класу ThreeColorGameObject .Різниця полягає в тому, що вам слід додати додаткову змінну-члену, яка вказує на те, чи готується м'яч на даний момент:

```
функція Ball () {
ThreeColorGameObject.call (це, sprites.ball_red, sprites.ball_green, sprites.ball_blue);
this.shooting = false; this.reset ();
}
Ball.prototype = Object.create (ThreeColorGameObject.prototype);
```

Коли створюється екземпляр Ball , вам потрібно викликати ThreeColorGameObject конструктор, так само, як і в класі Cannon .У цьому випадку ви проходите вздовж кулі спрайтів як параметри. Крім того, вам потрібно вказати змінну зйомки початкове значення false , і ви скидаєте кулю, викликавши скидання метод

Клас Ball чітко ілюструє, що відбувається, коли ви успадковуєте від іншого класу.Кожен примірник балу складається з частини, успадкованої від ThreeColorGameObject, та частини, яка визначена у класі Ball .На малюнку 4.2 показано, як виглядає пам'ять для об'єкта Ball без використання спадщини.На малюнку 4.3 також показано екземпляр Ball , але використовується механізм успадкування, введений у цьому глава

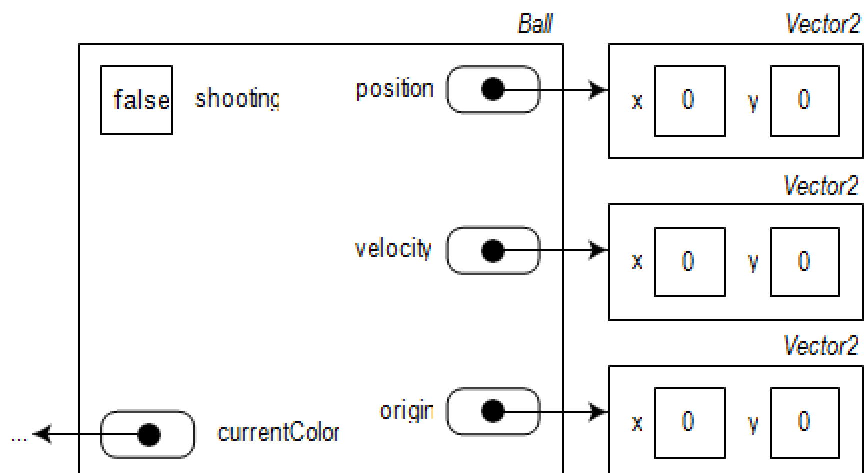


Рисунок 4.2 - *Огляд пам'яті, використовуваної примірником класу* ***Ball*** *(без спадщини)*

Куля

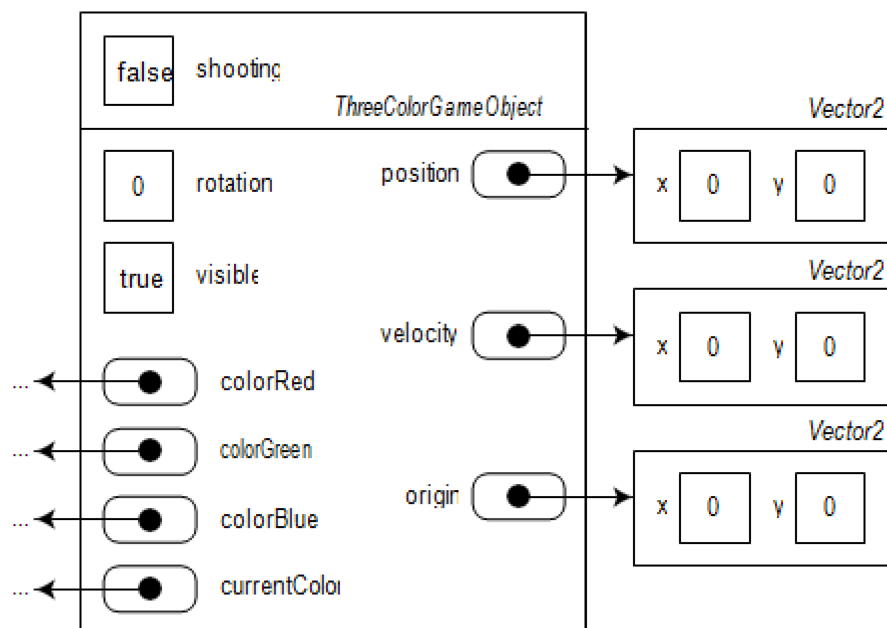


Рисунок 4.3 - *Примірник класу* ***кульки*** *(який успадковує від* ***ThreeColorGameObject)***

Ви можете трохи заплутати ці дві фігури та структури, які вони представляють.Пізніше в цьому розділі розглядається структура пам'яті трохи докладніше.На даний момент, припустимо , що складні об'єкти , що складаються з декількох змінних - членів (наприклад, Cannon або Ball

примірників) зберігаються інакше , ніж прості числа або булеві.Що це означає і як ви повинні правильно боротися з ним у своєму коді, наведено в кінці глава

Метод update в класі ThreeColorGameObject містить лише одну лінію коду, яка обчислює нову позицію ігрового об'єкта на основі його швидкості, часу проходження та поточної позиції:

```
this.position.addTo(this.velocity.multiply(delta));
```

Кулі повинні робити більше, ніж це. Швидкість руху повинна бути оновлена, щоб враховувати переміщення та сила тяжіння; колір м'яча повинен бути оновлений, якщо це необхідно; і якщо м'яч летить поза екраном, він повинен бути скинутий у вихідне положення. Ви можете просто скопіювати метод update з Попередній

версію класу Ball, щоб він замінив метод update ThreeColorGameObject .Трохи краще це зробити, це визначити метод update в класі Ball, але повторно використовувати оригінальний метод updateя з ThreeColorGameObject .Це можна зробити, використовуючи метод call, таким чином, дуже схожим на те, як ви використовували його для виклику конструктора суперкласу.Ось нова версія Методу Ball.update :

```
Ball.prototype.update = function (delta)
{ ThreeColorGameObject.prototype.update.call(this, delta); if (this.shooting) {
this.velocity.x *= 0.99;
this.velocity.y += 6;
}
else {
this.color = Game.gameWorld.cannon.color; this.position =
Game.gameWorld.cannon.ballPosition
.subtractFrom(this.center);
}
if (Game.gameWorld.isOutsideWorld(this.position)) this.reset();
};
```

Подивіться на першу інструкцію в цьому методі. Ви отримуєте доступ до об'єкта prototype ThreeColorGameObject , який містить функцію update .Ви викликаєте цю функцію update при передачі this об'єкта, тому об'єкт Ball оновлюється, але відповідно до методу update, визначеного в ThreeColorGameObject .Нарешті, ви передаєте параметр delta для цього дзвінка.Приємно, що цей підхід дозволяє окремо розділити (у цьому випадку) процес оновлення. Для будь-якого ігрового об'єкта з позицією та швидкістю потрібно оновити свою позицію, спираючись на її швидкість у кожній ітерації ігрової петлі. Ви визначаєте цю поведінку в методі update ThreeColorGameObject, щоб ви могли повторно використовувати його для будь-якого класу, який успадковує від ThreeColorGameObject.

Поліморфізм

Через механізм успадкування ви не завжди повинні знати, до якого типу об'єкта вказує змінна. Розглянемо наступну декларацію та ініціалізацію:

```
var someKindOfGameObject = new Cannon();
```

І десь в коді ви робите це:

```
someKindOfGameObject.update(delta);
```

Тепер припустимо, ви змінюєте декларацію та ініціалізацію, як показано нижче:

```
var someKindOfGameObject = new Ball();
```

Вам потрібно змінити виклик на метод update?Ні, ти не маєш, тому що так

Методи ігрового циклу, що викликаються, визначаються в класі ThreeColorGameObject .Коли ви викликаєте метод update на змінній someKindOfGameObject , то не має значення, який об'єкт гри насправді посилається.Єдине, що важливо, це те, що метод update визначено і що він очікує одного параметра: час, що минув з часу останнього виклику для update .Оскільки інтерпретатор стежить за тим, який об'єкт він є, правильна версія методу update викликається автоматично.

Цей ефект називається *поліморфізм* , і це іноді дуже корисно.Поліморфізм дозволяє краще відокремити код. Припустимо, що ігрове підприємство хоче випустити продовження своєї гри. Наприклад, він може захотіти ввести кілька нових ворогів або навички, які гравець може навчитися. Компанія може надавати ці розширення як підкласи загальних класів Enemy та Skill.Фактичний ігровий код буде використовувати ці об'єкти, не знаючи, до якої конкретної навички чи ворога він має справу. Він просто називає методи, визначені в загальних класах.

Ієрархія класів

Ви бачили кілька прикладів у цій главі класів, успадкованих від класу базових ігрових об'єктів.Клас повинен успадковуватись від іншого класу лише тоді, коли відносини між цими двома класами можуть бути описані як "є свого роду". Наприклад: " Ball" є своєрідним hreeColorGameObject. Ви можете написати інший клас, який успадковує від класу Ball , наприклад, BouncingBall , який може бути спеціальною версією стандартного м'яча, який відскакує від контейнерів для фарби, а не лише стикається з ними.І ви можете зробити ще один клас BouncingElasticBall, який успадковує від BouncingBall , який є м'ячем, що деформується відповідно до його еластичних властивостей, коли він відбивається від фарби.Щоразу, коли ви успадковуєте від класу, ви отримуєте

дані (закодовані в змінних членів) і в поведінка (закодований в методи і властивості) від в база клас за безкоштовно

Комерційні ігри мають ієрархію класів різних ігрових об'єктів з різними рівнями.Повертаючись до прикладу моделювання трафіку на початку цього розділу, можна уявити дуже складну ієрархію всіх видів різних транспортних засобів.На малюнку 4.4 показано приклад такої ієрархії.Цифра використовує стрілки, щоб вказати співвідношення успадкування між заняття

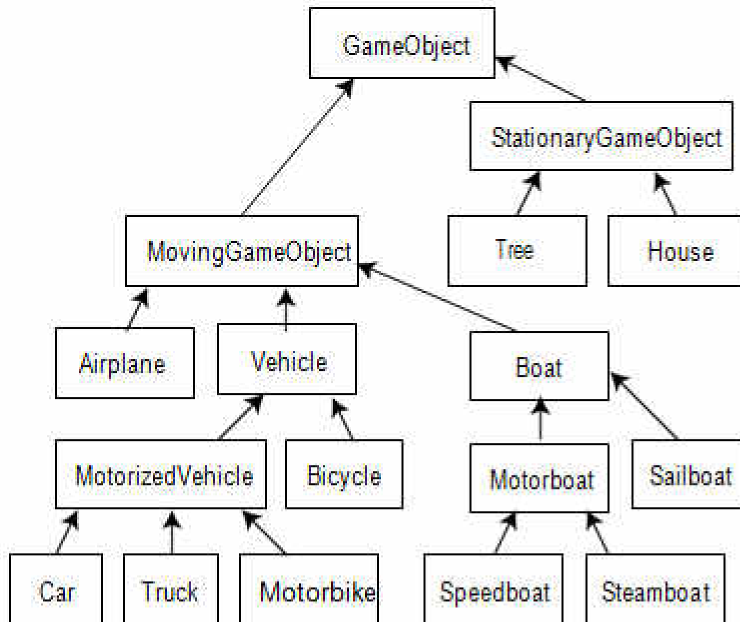


Рисунок 4.4 - *Комплексна ієрархія ігрових об'єктів в імітаційній грі*

На самому підставі дерева спадщини є клас GameObject .Цей клас містить лише дуже базову інформацію, таку як позиція або швидкість об'єкта гри. Для кожного підкласу можуть бути додані нові учасники (змінні, методи або властивості), які є релевантними для певного класу та його підкласів. Наприклад, змінна numberOfWheels зазвичай належить у класі Vehicle, а не у MovingGameObject (бо човен не має коліс).Перемінна flightAltitude належить в в Airplane клас і в змінна bellIsWorking належить Bicycle клас

Коли ви визначаєте, як складаються ваші класи, вам потрібно приймати багато рішень. Немає єдиної найкращої ієрархії; і, залежно від програми, одна ієрархія може бути більш корисною, ніж інша. Наприклад, цей приклад спочатку ділить клас MovingGameObject відповідно до середовища, який об'єкт використовує для зміни місця: землі, повітря чи води.Після цього класи поділяються в різних підкласах: моторизовані або не моторизовані.Ви могли б вибрати це зробити навпаки. Для деяких класів не цілком зрозуміло, де в ієрархії вони належать: чи ви кажете, що мотоцикл є особливим видом велосипеда (один з двигуном)? Або це особливий вид моторизованого транспортного засобу (один з двома колесами)?

Важливо те, що відносини між самими класами ясно. Вітрильник - це човен, але човен не завжди є вітрилом. Велосипед є транспортним засобом, але не кожен транспортний засіб - це велосипед.

Цінності та посилання

Перш ніж закінчити цю главу, давайте подивимося, як об'єкти та змінні розглядаються в пам'яті. При роботі з основними типами, такими як числа чи булеві, змінні безпосередньо пов'язані з місцем в пам'яті. Наприклад, подивіться на наступну декларацію та ініціалізацію:

```
var i = 12;
```

Після виконання цієї інструкції пам'ять виглядає так, як показано на Малюнку 4.5 .

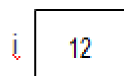


Рисунок 4.5 - *Використання пам'яті цифрової змінної*

Тепер ви можете створити нову змінну j і зберегти значення змінної i в цій змінній:

```
var j = i;
```

На малюнку 4.6 показано, як виглядає пам'ять після виконання цієї інструкції.

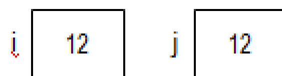


Рисунок 4.6 - *Використання пам'яті після оголошення та ініціалізації двох змінних номера*

Якщо ви призначите інше значення для j- змінної, наприклад, виконавши команду j = 24 , отримана пам'ять буде показана на малюнку 4.7 .

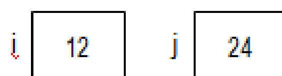


Рисунок 4.7 *Використання пам'яті після зміни значення* ***j-*** *змінної*

Тепер давайте подивимося, що відбувається, коли ви використовуєте змінні

більш складного типу, наприклад,

Cannon класу.Розглянемо наступний код:

```
var cannon1 = new Cannon();
var cannon2 = cannon1
```

Дивлячись на попередній приклад з використанням типу номера, можна очікувати, що в пам'яті є два об'єкти Cannon : один зберігається в змінній cannon1 , а один зберігається в cnon2 .Однак це не так! Насправді, як гармата1 , так і гармата2 *відносяться до одного і того ж об'єкта* .Після першої інструкції (створення об'єкта Cannon) пам'ять показана на малюнку 4.8 .

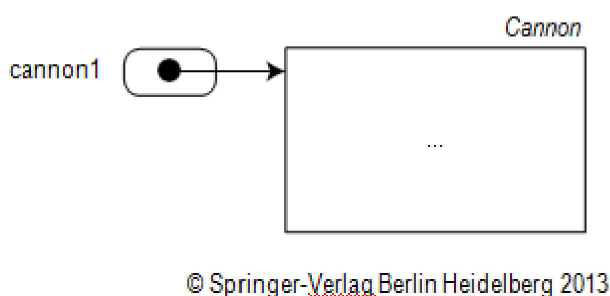


Рисунок 4.8 - *Об'єкт* ***Гармата*** *в пам'яті*

Тут ви бачите, що існує велика різниця між тим, як у пам'яті представлені основні типи, такі як числа та булевини, на відміну від більш складних типів, таких як клас Cannon .У JavaScript всі об'єкти, які не є примітивними типами, такі як числа, булевики та символи, зберігаються як *посилання,* а не значення.Це означає, що така змінна, як cannon1 , не містить безпосередньо об'єкта Cannon , але містить *посилання на неї* .На малюнку 11-8 вказується, що cannon1 є посиланням, представляючи його як блок, що містить стрілу до об'єкта.Якщо ви зараз оголошуєте змінну cannon2 і присвоюєте значення cannon1 Для цього ви можете побачити нову ситуацію на мал. 4.9 .

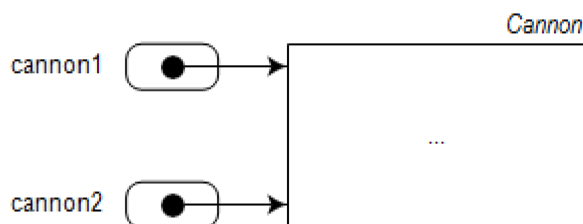


Рисунок 4.9 - *Дві змінні, що відносяться до одного і того ж об'єкта*

Результат полягає в тому, що якщо ви зміните колір гармати наступним чином

```
cannon2.color = Color.red;
```

Передача основних типів, таких як числа як параметрів для методів, відбувається *за значенням* , тому зміна значення в методі не має ефекту.Розглянемо наступне функція

```
function square(f) { f = f * f;
}
```

і тепер наступні інструкції:

```
var someNumber = 10; square(someNumber);
```

Після виконання цих інструкцій значення someNumber все ще становить 10 (а не 100).Чому це так? Оскільки, коли функція square викликається, параметр номера передає *значення* .Перемінна f є локальною змінною у методі, який спочатку містить значення змінної someNumber .У методі локальна змінна f змінюється так, щоб вона містила f * f , але це не змінює змінну someNumber , тому що це інше місце в пам'яті.Оскільки непримітивні об'єкти передаються за *посиланням* , наступний приклад призведе до зміни значення об'єкта, переданого як параметр:

```
function square(obj) { obj.f = obj.f * obj.f;
}

var myObject = { f : 10 }; square(myObject);
// myObject.f now contains the value 100.
```

Кожного разу, коли працює скрипт JavaScript, в пам'яті є цілий ряд посилань і значень .Наприклад, якщо ви подивитесь на рисунки 11-2 та 11-3 , ви бачите, що об'єкти Ball мають як значення, так і посилання на інші об'єкти (наприклад, об'єкти Vector2 або Image об'єкти).

Нуль та невизначеність

Кожного разу, коли ви оголошуєте змінну в JavaScript, спочатку його значення встановлено як невизначене :

```
var someVariable;
console.log(someVariable);  // will print 'undefined'.
```

У JavaScript ви також можете вказати, що змінна визначена, але в даний час не стосується жодного об'єкта. Це робиться за допомогою нульового ключового слова:

```
var anotherCannon = null;
```

Оскільки ви ще не створили об'єкт (використовуючи new ключове слово), пам'ять виглядає так, як показано на Малюнку 4.10 .

Рисунок 4.10 - *Перемінна, що вказує на* ***нуль***

Отже, вказавши, що змінна ще не вказує на щось, робиться шляхом присвоєння йому нуль .Можна навіть перевірити, чи є змінна до об'єкту чи ні, у програмі JavaScript, наприклад:

```
if (anotherCannon === null) anotherCannon = new Cannon();
```

У цьому прикладі ви перевіряєте, чи є змінна дорівнює null (не вказуючи на об'єкт).Якщо так, ви створюєте екземпляр Cannon, використовуючи new ключове слово, після чого ситуація в пам'яті змінився знову (побачити Малюнок 11-11).

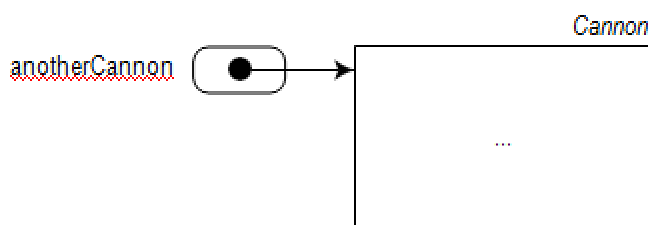


Рисунок 4.11 - *Остаточна ситуація в пам'яті*

Ви вирішите, коли ви хочете використовувати null і undefined .Не всі програмісти роблять це однаково. Ми пропонуємо вам використовувати undefined, щоб вказати, що змінної не існує, і null, щоб вказати, що ця змінна існує, але ще не стосується жодного об'єкта.

СЕМІНАРИ

Семінар 1
Створення ігор з використанням полотна та спрайтів

Анотація. Семінар орієнтований на отримання студентами знань та навичок роботи з елементом canvas, створення ігрового циклу, роботи з об'єктами, спрайтами та анімацією.

Мета семінару:

Мета семінару:

- Вивчити особливості елементу canvas.
- Вивчити поняття ігрового циклу та отримати навички роботи з ним.
- Вивчити поняття об'єктів, спрайтів.

У разі проходження семінару студент буде знати особливості роботи з елементом canvas, поняття ігрового циклу та отримає навички роботи з ним, вивчить поняття об'єктів, спрайтів.

1.1 Використання полотна

Елемент canvas був введений у HTML5 та забезпечує API для візуалізації в Інтернеті. API є простим, але якщо ви ніколи не виконували роботу з графікою, перш ніж це може зайняти деяккий час. API має чудову крос-браузерну підтримку на.

Використання canvas дуже просто. Треба лише створити тег <canvas>, створити контекст рендеринга з нього мовою javascript та використовуйте такі методи, як fillRect і drawImageконтекст, щоб відобразити форми та зображення. У API є багато методів для відтворення довільних шляхів, застосування перетворень тощо.

На цьому семінарі ми навчимося створювати 2D гру з полотном. Це буде справжня гра з спрайтми, анімаціями, виявленнями зіткнень і, звичайно, вибухами (рис. 1.1).

Рис.1.1. Скріншот гри

Почнемо розбиратися в коді. Більшість коду гри знаходиться в скрипті app.js в каталозі js.

Саме перше, що ми робимо, це створюємо тег полотна, встановлюємо ширину і висоту і додаємо його до тега body. Ми робимо це динамічно, щоб зберегти все в JavaScript, але ви можете додати тег canvas у файл HTML і використовувати щось подібне getElementById. Немає ніякої різниці між цими двома методами, це лише питання переваги, де створювати елемент canvas:

```
// Create the canvas
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
canvas.width = 512;
canvas.height = 480;
document.body.appendChild(canvas);
```

Елемент canvas має метод getContext, який є те, що ви використовуєте, щоб отримати контекст рендеринга. Контекст - це об'єкт, через який ви визиваєте всі рендерингові API. Ви також можете передати, webgl якщо ви хочете використовувати контекст WebGL для 3d сюжетів.

Звідси ми будемо використовувати змінну ctx.

1.2 Ігровий цикл

Вам потрібен ігровий цикл, який постійно оновлюється та ґарантує гру. Він виглядає так:

```
// The main game loop
var lastTime;
function main() {
        var now = Date.now();
        var dt = (now - lastTime) / 1000.0;

          update(dt);
          render();

          lastTime = now;
          requestAnimFrame(main);
};
```

Ви оновлюєте та відтворюєте сцену, а потім використовуєте requestAnimationFrame для наступного циклу. В основному це більш розумний спосіб сказати setTimeout(main, 1000/60), який намагається зробити 60 кадрів в секунду. У самому верху app.js ми використовуємо rAF як функцію requestAnimFrame, оскільки ще не всі браузери підтримують її.

Ніколи *не* використовуйте setTimeout(main, 1000/60), тому що це менш точно, а також витрачає багато циклів, коли це не є необхідним.

Функція update приймає час, який пройшов з моменту останнього оновлення. **Ніколи не** відновляйте свою сцену постійними значеннями на кадр (як x += 5;). Робота вашої гри буде відрізнятися на різних комп'ютерах і платформах, тому вам потрібно оновити свою сцену незалежно від частоти кадрів.

Це досягається шляхом обчислення часу з моменту останнього оновлення (у секундах) і вираження всіх рухів у пікселях/секундах. Рух потім стає x += 50 * dt, або "50 пікселів за секунду".

1.3 Завантаження ресурсів та запуск гри

Наступний розділ коду ініціалізує гру та завантажує всі ресурси. Він використовує один з небагатьох окремих класів: resources.js. Це дуже проста бібліотека, яка завантажує зображення та запускає подію, коли всі вони завантажені.

Ігри вимагають великої кількості ресурсів, таких як зображення, дані сцени тощо. Для 2D ігор більшість або всі ресурси - це зображення. Перед початком гри потрібно завантажити всі свої ресурси, щоб їх можна було негайно використовувати.

Легко завантажити зображення в javascript і зробити щось, коли це доступно:

```
var img = new Image();
img.onload = function() {
   startGame();
```

```
};
img.src = url;
```

Це стає дуже нудним, коли у вас є декілька зображень для завантаження. Вам потрібно зробити купу глобальних змінних, і кожен раз перевіряти, чи всі вони завантажені. Розглянемо базовий ресурсний завантажувач, щоб автоматично обробляти все це.

```
(function() {
var resourceCache = {};
var loading = [];
var readyCallbacks = [];

// Load an image url or an array of image urls
function load(urlOrArr) {
if(urlOrArr instanceofArray) {
        urlOrArr.forEach(function(url) {
            _load(url);
        });
    }
else {
        _load(urlOrArr);
    }
  }

function _load(url) {
if(resourceCache[url]) {
return resourceCache[url];
    }
else {
var img = new Image();
        img.onload = function() {
            resourceCache[url] = img;

if(isReady()) {
                readyCallbacks.forEach(function(func) { func(); });
            }
        };
        resourceCache[url] = false;
        img.src = url;
    }
  }

function get(url) {
```

```
return resourceCache[url];
    }

function isReady() {
var ready = true;
for(var k in resourceCache) {
if(resourceCache.hasOwnProperty(k) &&
              !resourceCache[k]) {
               ready = false;
            }
        }
return ready;
    }

function onReady(func) {
        readyCallbacks.push(func);
    }

window.resources = {
        load: load,
        get: get,
        onReady: onReady,
        isReady: isReady
    };
})();
```

Гра викликає resources.load з усіма зображеннями для завантаження, а потім викликає resources.onReady, щоб зареєструвати зворотний виклик, коли завантажено все. Це передбачає, що ви не будете викликати resources.load пізніше в грі; він працює тільки при запуску.

Він зберігає кеш-пам'ять зображень resourceCache, а при завантаженні зображення перевіряє, чи завантажені всі зображення, і, якщо так, викликає всі зареєстровані зворотні виклики. Тепер ми можемо просто зробити це в нашій грі:

```
resources.load([
'img/sprites.png',
'img/terrain.png'
]);
resources.onReady(init);
```

Щоб отримати зображення після початку гри, ми просто робимо resources.get('img/sprites.png'). Ви можете вручну завантажувати зображення та запускати гру або використовувати щось на зразок resources.js, щоб полегшити його.

У вищезгаданому коді init визивається, коли завантажуються всі зображення, що створює фоновий малюнок, фіксує кнопку «Відтворити знову», скидає стан гри та запускає гру.

```
function init() {
   terrainPattern = ctx.createPattern(resources.get('img/terrain.png'), 'repeat');

document.getElementById('play-again').addEventListener('click', function() {
      reset();
   });

   reset();
   lastTime = Date.now();
main();
}
```

1.4 Стан гри

Почнемо реалізовувати певну логіку гри. В основі кожної гри лежить *стан гри* . Це дані, які відображають поточний стан: список об'єктів на сцені з позицією та іншою інформацією, поточну оцінку, час після останнього звільнення гравця та все інше:

```
// Game state
var player = {
   pos: [0, 0],
   sprite: new Sprite('img/sprites.png', [0, 0], [39, 39], 16, [0, 1])
};

var bullets = [];
var enemies = [];
var explosions = [];

var lastFire = Date.now();
var gameTime = 0;
var isGameOver;
var terrainPattern;

// The score
var score = 0;
var scoreEl = document.getElementById('score');
```

Більшість змінних відстежує, коли гравець в останнє випустив кулю (lastFire), як довго продовжується гра (gameTime), якщо гра закінчилася (isGameOver), зображення зображень (terrainPattern) та кількість балів

(score). Існує також список об'єктів на сцені: кулі, вороги та вибухи.

Існує також сутність player, яка стежить за тим, де знаходиться гравець, і за станом спрайту. Перш ніж перейти до коду, давайте поговоримо про сутності і спрайти.

1.5 Сутності та спрайти

Сутності

"Сутність" є об'єктом на сцені. Все, від корабля до кулі, є сутністю.

Сутність у цій системі - це прості JavaScript-об'єкт, який відстежує, де він знаходиться на сцені та багато іншого. Це досить проста система, в якій ми вручну обробляємо кожний тип сутності. Кожен із наших сутностей має поля posі і sprite, і можливо більше. Наприклад, якщо ми хочемо додати ворога на нашу арену, ми зробимо це таким чином:

```
enemies.push({
   pos: [100, 50],
   sprite: new Sprite(/* sprite parameters */)
});
```

Це додає ворога в позицію x = 100 і y = 50 з вказаним спрайтом.

Спрайти та анімація

Спрайт ("Sprite") - це зображення, яке відображається для представлення об'єкта. Спрайти є складнішими, тому що ми хочемо анімувати їх. Без анімації спрайти можуть бути простими зображеннями, які відображаються за допомогою ctx.drawImage.

Ми можемо реалізовувати анімацію, завантажуючи декілька зображень і просуваючи їх через час. Це називається анімацією за ключовими кадрами.

Щоб полегшити редагування кожного ключового кадру та завантаження його, ці зображення зазвичай надсилаються в єдине зображення, яке називається картою спрайтів. Ви, можливо, вже знайомі з цією технікою в CSS . Фактично, багато разів *кілька* різних анімацій спрайтів містяться в одній карті спрайтів.

Будемо використовувати жорсткий набір графіки, який є простою групою bmp-файлів. Для цього скопіюємо потрібну індивідуальну графіку та вставимо їх у єдиний аркуш спрайтів. Для цього потрібен простий графічний редактор (все, що може рухати пікселі, повинно працювати).

Однак було б важко керувати всією цією анімацією вручну. Для цього є клас: sprite.js . Це невеликий файл, який розгортає логіку анімації у тип для багаторазового використання:

```
function Sprite(url, pos, size, speed, frames, dir, once) {
        this.pos = pos;
```

```
            this.size = size;
            this.speed = typeof speed === 'number' ? speed : 0;
            this.frames = frames;
            this._index = 0;
            this.url = url;
            this.dir = dir || 'horizontal';
            this.once = once;
    };
```

Це конструктор для класу Spritey. Він вимагає чимало аргументів, але не всі вони є обов'язковими. Давайте розглянемо їх по одному:

- url: шлях до зображення спрайта
- pos: координата x і y зображення для спрайту
- size: розмір спрайта (тільки один ключовий кадр)
- speed: швидкість кадрів / сек для анімації
- frames: масив покажчиків кадрів для анімації: [0, 1, 2, 1]
- dir: в якому напрямку рухати спрайт на карті: "горизонтальний" (за замовчуванням) або "вертикальний"

Для аргументу frames може знадобитися додаткове пояснення. Система передбачає, що всі кадри анімації однакового розміру. Під час анімації система просто проходить через карту спрайтів або горизонтально, або вертикально (залежно від dir), починаючи з pos і збільшуючи значення x або y.

Лише url, pos і size потрібні, оскільки вам не потрібна анімація.

Кожен об'єкт Sprite має метод update, який оновлює анімаціюКожен спрайт повинен бути оновлений з кожним кадром.

```
Sprite.prototype.update = function(dt) {
this._index += this.speed*dt;
}
```

Кожен об'єкт Sprite також має метод render, який дозволяє фактично намалювати себе. Він перевіряє, щоб побачити, який кадр він повинен відобразити, обчислює координати на карті спрайтів та викликає ctx.drawImage щоб намалювати його.

```
Sprite.prototype.render = function(ctx) {
var frame;

if(this.speed>0) {
var max = this.frames.length;
var idx = Math.floor(this._index);
      frame = this.frames[idx % max];

if(this.once && idx >= max) {
```

```
this.done = true;
return;
        }
    }
else {
        frame = 0;
    }


var x = this.pos[0];
var y = this.pos[1];

if(this.dir == 'vertical') {
        y += frame * this.size[1];
    }
else {
        x += frame * this.size[0];
    }

    ctx.drawImage(resources.get(this.url),
                x, y,
this.size[0], this.size[1],
0, 0,
this.size[0], this.size[1]);
}
```

1.6 Оновлення сцени

Нам потрібно визначити функцію, яка обробляє ввод, оновлює всі спрайти, оновлюєи позиції об'єктів та керує зіткненнями.

```
function update(dt) {
    gameTime += dt;

    handleInput(dt);
    updateEntities(dt);

// It gets harder over time by adding enemies using this
// equation: 1-.993^gameTime
if(Math.random() <1 - Math.pow(.993, gameTime)) {
        enemies.push({
            pos: [canvas.width,
Math.random() * (canvas.height - 39)],
            sprite: new Sprite('img/sprites.png', [0, 78], [80, 39],
6, [0, 1, 2, 3, 2, 1])
        });
```

```
    }

    checkCollisions();

    scoreEl.innerHTML = score;
};
```

Зверніть увагу, як ми додаємо нових ворогів на сцену. Ми додаємо ворога, якщо довільне значення нижче, ніж порогове, і ворог додається в правий бік гри поруч із представленням. Він випадковим чином розміщується на осі шляхом множення випадкового значення на висоту полотна, за винятком висоти противника, так що нижня частина не зрізається.

Вхідні дані

Для обробки вхідних даних створимо ще одну бібліотеку: input.js . Це дуже маленька бібліотека, яка просто зберігає стан натисканих клавіш, додавши в документ оброблювачі подій keydown і keyup.

Бібліотека експортує одну функцію, input.isDown. Тепер ми можемо обробляти дані таким чином:

```
function handleInput(dt) {
if(input.isDown('DOWN') || input.isDown('s')) {
        player.pos[1] += playerSpeed * dt;
    }

if(input.isDown('UP'|| span>) || input.isDown('w')) {
        player.pos[1] -= playerSpeed * dt;
    }

if(input.isDown('LEFT') || input.isDown('a')) {
        player.pos[0] -= playerSpeed * dt;
    }

if(input.isDown('RIGHT') || input.isDown('d')) {
        player.pos[0] += playerSpeed * dt;
    }

if(input.isDown('SPACE') &&
       !isGameOver &&
Date.now() - lastFire >100) {
var x = player.pos[0] + player.sprite.size[0] / 2;
var y = player.pos[1] + player.sprite.size[1] / 2;

        bullets.push({ pos: [x, y],
```

```
                dir: 'forward',
                sprite: new Sprite('img/sprites.png', [0, 39], [18, 8]) });
        bullets.push({ pos: [x, y],
                dir: 'up',
                sprite: new Sprite('img/sprites.png', [0, 50], [9, 5]) });
        bullets.push({ pos: [x, y],
                dir: 'down',
                sprite: new Sprite('img/sprites.png', [0, 60], [9, 5]) });


        lastFire = Date.now();
      }
    }
```

Якщо користувач натискає стрілку вниз або клавішу 's', ми рухаємо гравця вгору по осі y. Система координат полотна розміщує (0, 0) у лівому верхньому кутку, тому рух вгору по осі y переміщує об'єкт вниз по екрану. Ми робимо те ж саме для клавіш вгору, вліво та вправо.

Зверніть увагу, що ми визначили змінну playerSpeed у верхній частині програми app.js . Ось визначені нами швидкості:

```
// Speed in pixels per second
var playerSpeed = 200;
var bulletSpeed = 500;
var enemySpeed = 100;
```

Помноживши playerSpeed на параметр dt, ми обчислимо правильну кількість пікселів для переміщення. Якщо після останнього оновлення (параметра dt) пройшла 1 секунда, гравець буде переміщати 200 пікселів. Якщо минуло 5 секунд, він перемістить 100 пікселів. Це показує постійну швидкість руху, незалежно від частоти кадрів.

Останнє, що ми робимо - постріл кулі, якщо натиснути клавішу пробілу, гра не закінчилася. lastFire це глобальна змінна, яка є частиною стану гри. Це допомагає нам контролювати швидкість пострілів; інакше гравець може запустити кулю в *кожному кадрі:*

```
var x = player.pos[0] + player.sprite.size[0] / 2;
var y = player.pos[1] + player.sprite.size[1] / 2;

bullets.push({ pos: [x, y],
        dir: 'forward',
        sprite: new Sprite('img/sprites.png', [0, 39], [18, 8]) });
bullets.push({ pos: [x, y],
        dir: 'up',
        sprite: new Sprite('img/sprites.png', [0, 50], [9, 5]) });
```

```
bullets.push({ pos: [x, y],
            dir: 'down',
            sprite: new Sprite('img/sprites.png', [0, 60], [9, 5]) });

lastFire = Date.now();
```

Об'єкти

Всі об'єкти потрібно оновити. У нас є одиночний об'єкт player і 3 масиви для куль, ворогів та вибухів.

```
function updateEntities(dt) {
// Update the player sprite animation
   player.sprite.update(dt);

// Update all the bullets
for(var i=0; i<bullets.length; i++) {
var bullet = bullets[i];

switch(bullet.dir) {
case'up': bullet.pos[1] -= bulletSpeed * dt; break;
case'down': bullet.pos[1] += bulletSpeed * dt; break;
default:
         bullet.pos[0] += bulletSpeed * dt;
      }

// Remove the bullet if it goes offscreen
if(bullet.pos[1] <0 || bullet.pos[1] > canvas.height ||
        bullet.pos[0] > canvas.width) {
         bullets.splice(i, 1);
         i--;
      }
   }

// Update all the enemies
for(var i=0; i<enemies.length; i++) {
      enemies[i].pos[0] -= enemySpeed * dt;
      enemies[i].sprite.update(dt);

// Remove if offscreen
if(enemies[i].pos[0] + enemies[i].sprite.size[0] <0) {
         enemies.splice(i, 1);
         i--;
      }
   }
```

```
// Update all the explosions
for(var i=0; i<explosions.length; i++) {
      explosions[i].sprite.update(dt);

// Remove if animation is done
if(explosions[i].sprite.done) {
         explosions.splice(i, 1);
         i--;
      }
   }
}
```

Почнемоз в ершини: спрайт гравця оновлюється, просто викликаючи функцію спрайту update. Це переміщує анімацію вперед.

Наступні 3 wbrkf проходять через окремі кулі, ворогів та вибухи. Рух кулі є найбільш складним:

```
switch(bullet.dir) {
case'up': bullet.pos[1] -= bulletSpeed * dt; break;
case'down': bullet.pos[1] += bulletSpeed * dt; break;
default:
   bullet.pos[0] += bulletSpeed * dt;
}
```

Якщо bullet.dir це 'up', ми рухаємо кулю вниз по осі Y. 'down' - рухатися вперед вздовж осі x.

```
// Remove the bullet if it goes offscreen
if(bullet.pos[1] <0 || bullet.pos[1] > canvas.height ||
  bullet.pos[0] > canvas.width) {
   bullets.splice(i, 1);
i--;
}
```

1.7 Виявлення зіткнень

Обробка зіткнень означає, що ви переміщуєте одиеабо обидва об'єкти так, що вони більше не стикаються. Існує 3 типи зіткнень, які потрібно перевірити:

1. Враги страждають від куль
2. Гравець вражає ворога
3. Гравець потрапляє на край екрана

Виявлення 2D зіткнень є простим:

```
function collides(x, y, r, b, x2, y2, r2, b2) {
```

```
return !(r <= x2 || x > r2 ||
        b <= y2 || y > b2);
}

function boxCollides(pos, size, pos2, size2) {
return collides(pos[0], pos[1],
            pos[0] + size[0], pos[1] + size[1],
            pos2[0], pos2[1],
            pos2[0] + size2[0], pos2[1] + size2[1]);
}
```

Семінар 2
Додаткові можливості використання ігрових циклів

Анотація. Семінар орієнтований на отримання студентами знань про написання ефективних основних циклів ігор.

Мета семінару:

- Освоїти прийоми створення ефективних основних циклів.
- Освоїти прийоми додавання елементів випадковості в ігрові веб-додатки.

У разі проходження семінару студент ознайомиться з прийомами створення ефективних основних циклів ігор.

Основний цикл є основною частиною будь-якої програми, в якому стан змінюється з плином часу. У іграх основний цикл часто відповідальний за обчислення фізики та AI, а також для виведення результату на екран.

2.1 Перша спроба

Ми збираємося написати "гру". Для простоти, ми просто намалюємо динамічну коробку:

```
<div id = "box" >< / div>
```

Давайте покажемо це:

```
#box {
   background-color: red;
   height: 50px;
   left: 150px;
   position: absolute;
   top: 10px;
```

```
    width: 50px;
}
```

Тепер давайте розглянемо скрипт. По-перше, наша коробка потребує деяких властивостей для визначення її положення та швидкості. Давайте також створимоо функцію draw(), щоб відобразити нову позицію вікна:

```
var box = document.getElementById('box'), // the box
    boxPos = 10, // the box's position
    boxVelocity = 2, // the box's velocity
    limit = 300; // how far the box can go before it switches direction

function draw() {
    box.style.left = boxPos + 'px';
}
```

Ми хочемо, щоб коробка рухалась вперед і назад, тому ми просто додамо швидкість до позиції.

```
function update() {
    boxPos += boxVelocity;
    // Switch directions if we go too far
    if (boxPos >= limit || boxPos <= 0) boxVelocity = -boxVelocity;
}
```

А тепер давайте запустимо. Для цього нам потрібен цикл, який просто виконує update() функцію, щоб перемістити вікно, а потім - draw() функцію оновлення візуалізації на екрані. Як ми повинні це зробити?

```
while ( true ) {
    update ( ) ;
    малювати ( ) ;
}
```

Проте, JavaScript є однопоточною, що означає, що цей підхід забороняє браузеру робити майже що інше на цій сторінці. В результаті браузер заблокується, і через кілька секунд він покаже користувачеві помилку. Потрібно, щоб гра працювала більше, ніж на кілька секунд.

Якщо ви знайомі з JavaScript, ви можете подумати про функції setTimeout() або setInterval(), які дозволяють виконувати код після певної кількості часу. Це досить розумна ідея, але в браузерах є кращий спосіб: requestAnimationFrame(). Це досить нова функція з підтримкою браузера. Ви викликаєте цю функцію, і він виконує цей зворотній виклик, коли веб-браузер готовий змінити зовнішній вигляд сторінки.

Гаразд, давайте використати requestAnimationFrame () для написання

цього основного циклу!

```
function mainLoop() {
   update();
   draw();
   requestAnimationFrame(mainLoop);
}

// Start things off
requestAnimationFrame(mainLoop);
```

Дуже важливо, що draw () викликається після update (), тому що ми хочемо, щоб на екрані відобразився стан програми максимально актуальним.

Поки що наша функція update () залежить від частоти кадрів. Іншими словами, якщо ваша гра працює повільно (тобто менше кадрів в секунду), ваш об'єкт також буде рухатися повільно, тоді як якщо ваша програма швидко працює (тобто більше кадрів в секунду), ваш об'єкт буде швидко рухатися. Наявність таких непередбачуваних ігрових процесів небажана, особливо в багатокористувацьких іграх. Ніхто не хоче, щоб їхній персонаж був повільним, адже їхній комп'ютер не такий потужний. Навіть у однокористувальницьких іграх швидкість значно впливає на складність: ігри, які потребують швидкої реакції, будуть простішими на повільних швидкостях і незручно важко на швидких.

Ми будемо використовувати той факт, що requestAnimationFrame() передає часовий покажчик на зворотний виклик. Кожного разу, коли цикл працює, ми перевіримо, чи пройшов мінімальний час; якщо він буде, ми будемо рендерити кадр, і якщо цього не зробить, ми просто чекаємо наступного кадру.

```
var lastFrameTimeMs = 0, // The last time the loop was run
   maxFPS = 10; // The maximum FPS we want to allow

function mainLoop(timestamp) {
   // Throttle the frame rate.
   if (timestamp < lastFrameTimeMs + (1000 / maxFPS)) {
      requestAnimationFrame(mainLoop);
      return;
   }
   lastFrameTimeMs = timestamp;

   // ...
}
```

Проблема в тому, що наша програма не прив'язана до реального часу ... як ми це виправляємо?

Давайте спробуємо помножити швидкість на кількість часу, що пройшло між кадрами рендеринга, та передати це значення delta. Тому замість запуску boxPos += boxVelocity ми запустимо boxPos += boxVelocity * delta. Нам потрібно буде налаштувати нашу функцію оновлення, щоб отримати delta як параметр з основного циклу:

```
// Re-adjust the velocity now that it's not dependent on FPS
var boxVelocity = 0.08,
   delta = 0;

function update(delta) { // new delta parameter
   boxPos += boxVelocity * delta; // velocity is now time-sensitive
   // ...
}

function mainLoop(timestamp) {
   // ...

   delta = timestamp - lastFrameTimeMs; // get the delta time since last frame
   lastFrameTimeMs = timestamp;

   update(delta); // pass delta to update
   // ...
}
```

Семінар 3
Художнє оформлення ігрового веб-додатка мовою JavaScript

Анотація. Семінар орієнтований на отримання студентами навичок додавання ефектів руху, звуків, музики до ігрового додатка.

Мета семінару:

- Освоїти прийоми роботи додавання ефекту руху до ігрових об’єктів.
- Навчитися додавати звук та музику до ігрових додатків.

У разі успішного проходження практичного заняття студент буде знати основні прийоми додавання ефектів руху, звуків та музики.

3.1. Додавання ефектів руху

Щоб зробити гру більш візуально привабливою, ви можете ввести гарний поворотний ефект у рух банок з красою, щоб імітувати вплив вітру та тертя на падіння руху. Додавання такого ефекту не є складним. До методу update класу PaintCan потрібно додати лише одну лінію. Оскільки PaintCan - це підклас ThreeColorGameObject , він вже має rotation членна змінна, яка автоматично враховується, коли спрайт намальований на екран!

Для досягнення ефекту руху ви використовуєте метод Math.sin .Якщо значення залежить від поточної позиції банку, ви отримуєте різні значення залежно від цієї позиції. Потім ви використовуєте це значення, щоб застосувати *обертання* на справі.Це рядок коду, який ви додаєте до методу PaintCan.update :

```
this.rotation = Math.sin(this.position.y / 50) * 0.05;
```

Ця інструкція використовує y-координату позиції paint-can, щоб отримати різні значення обертання. Крім того, ви розділите його на 50, щоб отримати приємний повільний рух; і ви помножте результат на 0,05, щоб зменшити амплітуду синуса, тому обертання виглядає більш-менш реалістично. Якщо вам подобається, ви можете спробувати різні цінності та подивитися, як вони впливають на поведінку фарбників.

Навіть якщо ти є ні ан художник це допомагає до Бути здатний до зробити простий спрайты самі Це дає можливість ви до швидко зробити а прототип в гра-і може бути знайти вийшов там також є ан художник всередині ви. До створити спрайти ви перший треба добре інструменти Більшість художників використовувати а фарбування програма люблю Adobe Photoshop або а вектор малюнок програма люблю Adobe Ілюстратор але інші працювати з такий простий інструменти як Microsoft Фарба або в більше великий і безкоштовно GIMP Кожен інструмент вимагає практика Робота твій шлях через дещо підручники і зробити впевнений ви отримати дещо в поле зору в в багато хто інший особливості Часто в речі ти хочеш може Бути досягнуто в ан легко шлях

Переважно, створити дуже великий зображення від твій гра об'єкти і потім масштаб їх вниз до в вимагається розмір The переваги є це ви може змінити в вимагається розмір в твій гра пізніше і це ви отримати позбутися від псевдонім ефекти за рахунок до зображення

буття представлена по пікселів Коли масштабування зображення згладжування техніки суміш в кольори так в зображення залишається гладким Якщо ви тримати в назовні від в гра об'єкт в в зображення прозорий потім, коли ви масштаб в кордон пікселів буде автоматично стати частково прозорий Тільки якщо ви хочеш до створити в класичний піксель стиль повинен ви створити в спрайт у фактичному розмірі вимагається.

Нарешті, дивись навколо на в Веб Там є багато від спрайты це ви може

використовувати за безкоштовно Зробити впевнений до перевірити в ліцензія терміни так що упаковка від спрайты ти є використовуючи є законний за що ти є будівля ви може потім використовувати їх як а основи за твій власний спрайтыАле в в кінець усвідомити це в якість від твій гра збільшується значно коли ви робота з ан досвідчений художник

Додавання звуків і музики

Інший спосіб зробити гру більш приємною - це додавати звук. Ця гра використовує як фонову музику, так і звукові ефекти. Щоб у JavaScript було трохи полегшено прослуховування звуків, ви додаєте клас Sound, який дозволяє відтворювати звукові сигнали та звучати циклічно.Ось конструктор цього класу:

```
function Sound(sound, looping) {
this.looping = typeof looping !== 'undefined' ? looping : false; this.snd = new Audio();
if (this.snd.canPlayType("audio/ogg")) { this.snd.src = sound + ".ogg";
} else if (this.snd.canPlayType("audio/mpeg")) { this.snd.src = sound + ".mp3";
} else // we cannot play audio in this browser this.snd = null;
}
```

Оскільки не всі веб-переглядачі здатні відтворювати всі види музики, ви додали команду if, яка завантажує різні типи звуків залежно від того, який тип браузера може відтворювати.Подібно створенню об'єктів Image (для представлення спрацьовування) ви створюєте Audio та ініціалізуєте його джерело у звуковий файл, який потрібно завантажити.Крім звукового файла, ви додаєте змінну looping, яка вказує на те, чи повинен звук бути циклічним.Загалом, фонову музику слід петлювати; звукові ефекти (наприклад, стрільба фарбою) не слід.

Крім конструктора, ви додаєте метод під назвою play .У цьому методі ви завантажуєте звук, і ви встановлюєте атрибут, що називається autoplay, на значення true.Результатом цього є те , що звук буде відразу ж почати грати після того, як він завантажений.Якщо звук не потрібен, ви закінчите і можете повернутися з методу. Якщо вам потрібен звук для циклу, вам доведеться перезавантажити та відтворити звук після його закінчення. Тип Audio дозволяє прикріплювати функції до так званих *подій* .Коли виникає подія, функція, яку ви додаєте, виконується. Прикладом може служити подія, за яким звук почав відтворюватися, або подія, в якій звук завершився грає

Ця книга використовує дуже мало подій та обробку подій. Хоча на них покладаються низку концепцій JavaScript. Наприклад, натискання клавіш і дії миші створюють події, які потрібно обробляти у ваших іграх. У цьому випадку ви бажаєте виконати функцію, коли звук завершився. Ось повний метод play:

```
Sound.prototype.play = function () { if (this.snd === null)
return; this.snd.load(); this.snd.autoplay = true; if (!this.looping)
return;
this.snd.addEventListener('ended', function () { this.load();
this.autoplay = true;
}, false);
};
```

Нарешті, ви додаєте властивість, щоб змінити гучність звуку, що відтворюється. Це особливо корисно, оскільки зазвичай ви хочете, щоб звукові ефекти були голоснішими, ніж фонову музику. У деяких іграх ці томи можуть бути змінені плеєром (пізніше в книзі ви бачите, як це зробити). Кожного разу, коли ви вводите звук у свою гру, обов'язково завжди надайте гучність або принаймні звук контролю. Ігри, які не мають змоги звучати звук, відчувають гнів роздратованих користувачів через відгуки! Ось власність обсягу , яка є прямо:

```
Object.defineProperty(Sound.prototype, "volume",
{
get: function () {
return this.snd.volume;
},
set: function (value) { this.snd.volume = value;
}
});
```

У файлі Painter.js , в якому ви завантажуєте всі активи, ви завантажуєте звуки та зберігаєте їх у змінній, як і для спрацьовування:

```
var sounds = {};
```

І ось, як ви завантажуєте відповідні звуки, використовуючи клас Sound, який ви щойно створили:

```
var loadSound = function (sound, looping) {
return new Sound("../../assets/Painter/sounds/" + sound, looping);
};

sounds.music = loadSound("snd_music");
sounds.collect_points = loadSound("snd_collect_points"); sounds.shoot_paint = loadSound("snd_shoot_paint");
```

Зараз дуже легко грати звуки під час гри. Наприклад, коли ініціалізується гра, ви починаєте відтворювати фонову музику з невеликим

об'ємом, як показано нижче:

```
sounds.music.volume = 0.3; sounds.music.play();
```

Ви також хочете відтворити звукові ефекти. Наприклад, коли гравець вистрілює м'яч, вони хочуть почути це! Отже, ви відтворюєте цей звуковий ефект, коли вони починають знімати м'яч. Це розглядається в методі handleInput класу Ball :

```
Ball.prototype.handleInput = function (delta) { if (Mouse.leftPressed && !this.shooting) {
this.shooting = true;
this.velocity = Mouse.position.subtract(this.position)
.multiplyWith(1.2); sounds.shoot_paint.play();
}
};
```

Подібним чином, ви також граєте звук, коли фарба з правильним кольором випадає з екрана.

Підтримка Оцінка

Оцінки часто є дуже ефективним способом мотивувати гравців продовжувати грати. *Високі оцінки* особливо добре працюють в цьому відношенні, оскільки вони вводять в грі конкурентний фактор: ви хочете стати краще, ніж AAA або XYZ (у багатьох ранніх аркадних іграх дозволено лише три символи для кожного імені в списку високого балу, що веде до дуже образні імена).Високі оцінки настільки мотиваційні, що існують сторонні системи, щоб включити їх в ігри. Ці системи дозволяють користувачам порівнювати себе з тисячами інших гравців у всьому світі. У Painter грі, ви тримаєте його простим і додати змінну score члена до класу PainterGameWorld , в якому для зберігання поточного рахунку:

```
function PainterGameWorld() { this.cannon = new Cannon(); this.ball = new Ball();
this.can1 = new PaintCan(450, Color.red); this.can2 = new PaintCan(575, Color.green); this.can3 = new PaintCan(700, Color.blue); this.score = 0;
this.lives = 5;
}
```

Гравець починає з нуля.Кожен раз, коли фарба може випасти за межі екрану, оцінка оновлюється. Якщо балон правильного кольору випадає з екрана, до балу додаються 10 балів. Якщо каньок не є правильним кольором, гравець втрачає а життя

Оцінка є частиною того, що називається *економікою* гри.Економіка гри в

основному описує різні витрати і заслуги в грі і як вони взаємодіють. Коли ви створюєте власну гру, завжди варто подумати про свою економіку. Що коштує речі і які переваги виконувати різні дії як гравець? І чи є ці дві речі в рівновазі один з одним?

Ви оновлюєте рахунок в класі PaintCan , де ви можете перевірити, чи може він вийти за межі екрана.Якщо так, то ви перевіряєте, чи має він правильний колір та оновлює оцінку та кількість гравців, що живуть відповідно. Потім ви перемістіть об'єкт PaintCan до вершини, щоб він знову падав.

```
if (Game.gameWorld.isOutsideWorld(this.position)) { if (this.color === this.targetColor) {
Game.gameWorld.score += 10; sounds.collect_points.play();
}
else
Game.gameWorld.lives -= 1; this.moveToTop();
}
```

Нарешті, коли кольорова палиця з потрібного кольору випадає з екрана, ви відтворюєте звук.

Більш повний клас Canvas2D

Окрім нанесення спрайтів на екран, ви також хочете накреслити поточний бал на екрані (інакше це не матиме сенсу зберігати його). До цього часу на полотні ви малювали лише зображення. Елемент HTML5 canvas також дозволяє малювати на ньому текст. Щоб намалювати текст, ви поширюєте клас Canvas2D_Singleton .

Поки ви модифікуєте клас малюнка полотна, ви також хочете зробити щось інше. Тепер, коли ви організували всі свої змінні в об'єкти, які можуть бути створені за допомогою класів, які можуть успадковуватись від інших класів, це чудовий момент, щоб подумати про те, яка інформація повинна бути змінена там. Наприклад, ви, мабуть, хочете лише змінити canvas та canvasContext

змінні в класі Canvas2D_Singleton .Вам не потрібно звертатися до цих змінних, наприклад, у класі Cannon .У класі Cannon ви хочете використовувати лише високий рівень поведінки, який надається методами в малюнку полотна клас

На жаль, у JavaScript немає способу безпосередньо контролювати доступ до змінних. Злий програміст міг написати десь у своїй програмі таку лінію коду:

```
Canvas2D.canvas = null;
```

Після виконання цього рядка коду нічого не може бути намальовано на екрані! Звичайно, ні один розумний програміст не напише щось подібне до цілей, але це гарна ідея, щоб максимально яко зрозуміти користувачам

ваших класів про те, які дані вони повинні змінювати та які дані є

внутрішніми для класу, і їх не слід змінювати. Один із способів зробити це - додати щось до назви будь-якої змінної, яка має бути внутрішньою. Ця книга додає підкреслення до всіх змінних, які є внутрішніми і не повинні

змінити поза класом, в якому вони входять.Наприклад, ось модифікований конструктор Canvas2D_Singleton клас, що слідує за цим правилом:

```
function Canvas2D_Singleton() { this._canvas = null; this._canvasContext = null;
}
```

Ви також додаєте новий метод до класу drawText , який можна використовувати для накреслення тексту на екрані в певній позиції.Метод drawText дуже схожий на метод drawImage .У обох випадках ви використовуєте контекстний тло для виконання перетворення перед нанесенням тексту. Це дозволяє малювати текст у будь-якому бажаному положенні на полотні. Крім того, ви можете змінити колір тексту та вирівнювання тексту (вліво, в центрі чи вправо).Подивіться на приклад Painter10, що належить до цього розділу, щоб побачити тіло цього метод

Використовуючи цей метод, тепер простіше малювати текст на екрані. Наприклад, це намальовано зеленим текстом у верхньому лівому куті екрана:

```
Canvas2D.drawText("Hello, how are you doing?", Vector2.zero, Color.green);
```

Символи і Рядки

Послідовність символів називається *рядком* на більшості мов програмування, включаючи JavaScript.Так само, як числа чи булеві праці, рядки є примітивними типами в JavaScript. Струни також *незмінні* .Це означає, що коли створити рядок, його неможливо змінити. Звичайно, все ще можна *замінити* рядок іншим рядком.Наприклад:

```
var name = "Patrick";
name = "Arjan";
```

У JavaScript рядки розмежовуються одиничними або подвійними символами цитування. Якщо ви починаєте рядок з подвійною цитатою, воно має закінчуватися подвійною цитатою. Отже, це не допускається:

```
var country = 'The Netherlands";
```

Коли ви призначаєте рядок для змінної, рядок називається *константою* .На додаток до рядкових значень, постійними значеннями можуть бути числа, логічні значення, undefined або null , як це виражається в синтаксичній діаграмі на мал. 12-1 .

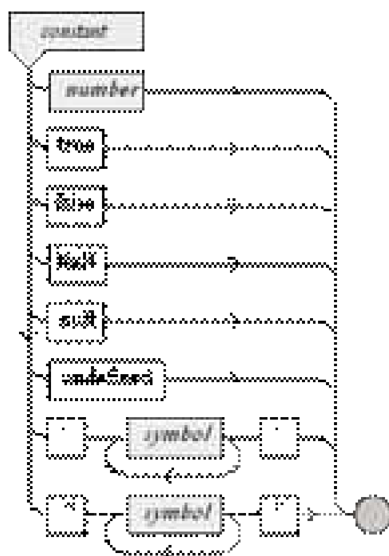


Рисунок 3.1 *- Синтаксична діаграма для постійних значень*

Використання одноразових та подвійних котирувань

Використовуючи рядкові значення та об'єднуючи їх з іншими змінами, ви повинні бути обережними, який тип цитат ви використовуєте (якщо таке є). Якщо ви забудете цитати, ви більше не пишете текст чи символи, а частину програми JavaScript! Існує велика різниця між ними

- Рядок “hello” і ім'я змінної hello
- Рядок '123' і значення 123
- Значення рядка '+' та оператора +

Спеціальний Символи

Спеціальні символи, просто тому, що вони є *особливими* , не завжди легко вказати використання одного символу між лапками.Отже, ряд спеціальних символів має спеціальні позначення, що використовують символ зворотної риски:

- '\ n' для закінчення рядка символ
- '\ t' для таблиць символ

Це вказує на нову проблему: як вказувати сам символ зворотнього слеша. Символ зворотнього коса рисунка позначається за допомогою *подвійної зворотної риски* .Аналогічним чином символ зворотнього смуги риски використовується для позначення символу для одиничних та подвійних лапок самі:

- '\\' для зворотної риски символ
- '\' ' або ' ' ' для єдиного символу котирування
- "\" " або " "" для символу подвійної цитати

Як ви може побачити, ви може використовувати сингл цитати без а

зворотна риска в а рядок розмежований по подвійний цитати, і пороку навпаки The синтаксис схема за представляючи все ці символи є дано в Малюнок 12-2 .

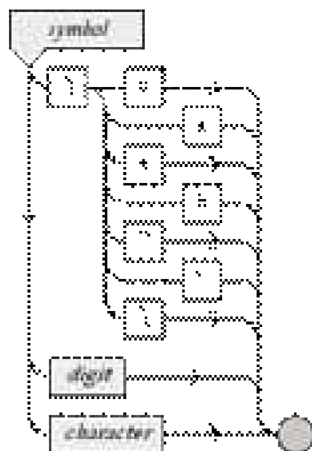


Рисунок 3.2 *- Синтаксична діаграма для символів*

Строкова операція

У грі Painter ви використовуєте рядкові значення в комбінації з методом drawText, щоб малювати текст певного кольору у потрібному шрифті десь на екрані.У цьому випадку ви хочете написати поточний рахунок у верхньому лівому кутку екрана. Оцінка зберігається в змінній-члені, що називається score . Це

змінна збільшена або зменшена в методі update PaintCan .Як ви можете побудувати текст, який слід надрукувати на екрані, оскільки частина цього тексту (оцінка) постійно змінюється? Рішення називається *конкатенацією рядків* , що означає скріплення одного фрагмента тексту за іншим.

У JavaScript (і на багатьох інших мовах програмування) це робиться за допомогою знака "плюс". Наприклад, вираз "Hi, my name is " + "Arjan" призводить до рядок Hi, my name is Arjan". .У цьому випадку ви об'єднаєте дві частини тексту. Це також можна зчепити фрагмент тексту і номера.Наприклад, вираз "Score: " + 200 призводить до рядок "Score: 200".Замість того, щоб використовувати константу, ви можете використовувати змінну. Отже, якщо змінна score містить значення 175, то вираз "Score: " + score оцінюється до "Score: 175"..Написавши це вираз як параметр методу drawText , ви завжди малюєте поточний бал на екрані.Остаточний виклик до drawText метод стає (див. PainterGameWorld клас)

```
Canvas2D.drawText("Score: " + this.score, new Vector2(20, 22),
Color.white);
```

Будьте уважні: конкатенація має сенс лише тоді, коли ви маєте справу з текстом. Наприклад, неможливо "об'єднати" два числа: вираз 1 + 2 дає 3 , а не 12 .Звичайно, ви можете об'єднати числа, *представлені як текст* : "1" + "2"

призводить до "12" .Визначення різниці між текстом і цифрами здійснюється за допомогою одиничних або подвійних лапок.

По суті, те, що таємно робиться в виразі "Score:" + 200, є *перетворення типу* або *відтворення* .

Чисельне значення 200 автоматично переноситься на рядок "200", перш ніж об'єднуватись до іншої рядки.

Якщо ви хочете перетворити рядове значення на чисельне значення, речі стають дещо складнішими.Це непроста операція для виконання інтерпретатора, оскільки не всі рядки можуть бути перетворені в чисельне значення. Для цього JavaScript має кілька корисних вбудованих функцій. Наприклад, як ви можете перетворити рядок у ціле число номер:

```
var x = parseInt("10");
```

Якщо рядка, передана як параметр, не є цілим числом, результат функції parseInt є цілою частиною числа:

```
var y = parseInt("3.14"); // y will contain the value 3
```

Щоб проаналізувати числа з десятками, JavaScript має функцію parseflow:

```
y = parseFloat("3.14"); // y will contain the value 3.14
```

Якщо рядок не містить дійсного числа, результат спроби розібрати його за допомогою однієї з цих двох функцій - це постійна NaN (не число, див. Також Рисунок 12-1).

Кілька остаточних зауважень

Вітаю - ти виконав свою першу гру! Малюнок 12-3 містить знімок екрана фінальної гри.Під час розробки цієї гри ви дізналися про багато важливих концепцій. У наступній грі ви розширюєте роботу, яку ви вже зробили. У той же час не забувайте грати в гру! Ви помітите, що це стає дійсно важко через кілька хвилин, тому що фарби банок знижуються швидше і швидше.

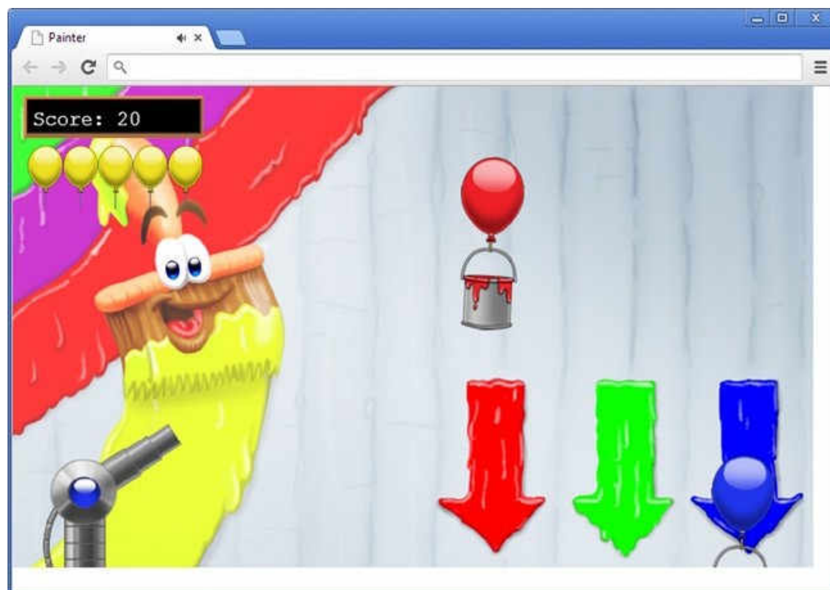


Рисунок 3.3 *- Знімок екрана остаточної версії Painter*

ЛІТЕРАТУРА

1. Ehhes A. Building JavaScript games for phones, tablets, and desktop. – Apress, 2014. – 410 p.
2. Spuy R. Advanced game design with HTML5 and JavaScript. – Apress, 2015. – 526 p.
3. Bunyan K. Build an YTML game: a developer’s guide with CSS and JavaScript. – San Francisco. – 220 p.
4. Brown E. Learning JavaScript. – O’Reilly Media. – 364 p.
5. Бахирев А.М. Сюрреализм на JavaScript. – Санкт-Петербуерг: СОНЭЛ, 2014. – 228 с.
6. Ehhes A. Building JavaScript games for phones, tablets, and desktop. – Apress, 2014. – 410 p.
7. Spuy R. Advanced game design with HTML5 and JavaScript. – Apress, 2015. – 526 p.
8. Bunyan K. Build an YTML game: a developer’s guide with CSS and JavaScript. – San Francisco. – 220 p.
9. Brown E. Learning JavaScript. – O’Reilly Media. – 364 p.
10. Бахирев А.М. Сюрреализм на JavaScript. – Санкт-Петербуерг: СОНЭЛ, 2014. – 228 с.

Навчальне електронне видання

ЖАРІКОВА М.В.

ЕЛЕКТРОННИЙ НАВЧАЛЬНИЙ ПОСІБНИК

«РОЗРОБКА ІГРОВИХ WEB-ДОДАТКІВ»

Для підготовки студентів
на першому (бакалаврському) рівні вищої освіти,
галузі знань 12 «Інформаційні технології»,
спеціальності 121 «Інженерія програмного забезпечення»,
освітньо-професійної програми «Програмна інженерія»

ISBN 978-617-7573-89-9 (електронне видання)

Підписано до видання 04.04.2019 р. Формат 60×84/4.
Гарнітура Times.
Ум. друк. арк. 23,02. Обл.-вид. акр. 24,75.
Замовлення № 1113.

Книжкове видавництво ФОП Вишемирський В. С.
Свідоцтво про внесення до Державного реєстру суб'єктів
видавничої справи: серія ХС № 48 від 14.04.2005 р.
видано Управлінням у справах преси та інформації
73000, Україна, м. Херсон, вул. Соборна, 2,
тел. (050) 133-10-13, e-mail: printvvs@gmail.com, vish_sveta@rambler.ru