**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Факультет комп’ютерних наук та кібернетики

Кафедра математичних основ комп’ютерних наук

**Курсова робота**

за спеціальністю 122 Комп’ютерні науки

на тему:

**Розробка бібліотеки симуляції поведінки твердих тіл**

Виконав студент 1 курсу

Єфремов Микола Сергійович

(підпис)

Науковий керівник:

асистент

Коваль Юрій Віталійович

(підпис)

Засвідчую, що в цій курсовій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент (підпис)

**ЗМІСТ**

Перелік використаних термінів та скорочень…………………………………………...3

Вступ……………………………………………………………………………………….4

Розділ 1. Фізичні рушії……………………………………………………………………5

1.1 Історія виникнення…………………………………………………………………….5

1.2 Застосування фізичних рушіїв………………………………………………………..6

1.3 Типи реалізації поведінки об’єктів у 2D платформерах……………………………7

Розділ 2. Стадії роботи фізичного рушія………………………………………………...11

2.1 Визначення колізій…………………………………………………………………....11

2.2 Представлення колізій………………………………………………………………..12

Розділ 3. Порівнняня існуючих фізичних рушіїв……………………………………….17

Godot……………………………………………………………………………………….17

Box2D………………………………………………………………………………………18

Cocos2D…………………………………………………………………………………....19

Coherent UI………………………………………………………………………………...20

Розділ 4. Розробка…………………………………………………………………………21

4.1 Опис фізічних законів………………………………………………………………...21

4.2 Реалізація collision detection…………………………………………………….……23

Висновки…………………………………………………………………………………..24

Список літератури та перелік джерел посилання…………………………..25

**ПЕРЕЛІК ВИКОРИСТАНИХ ТЕРМІНІВ ТА СКОРОЧЕНЬ**

**AABB -** бмежуючий паралелепіпед вирівняний по координатним осях) - це обмежена область в просторі у вигляді прямокутного паралелепіпеда, зі сторонами, паралельними осям координат у світовій системі. При обертанні об'єкта AABB змінює свої розміри, але завжди залишається орієнтованим по осях координат. Другий варіант AABB - використовувати в якості Bounding box'а куб зі сторонами рівними максимального розміру об'єкта. Таким чином, в будь-якому положенні об'єкт не вийде за межі AABB, хоча, звичайно, при цьому втрачається точність.

**Тайл**— зображення, яке використовується для створення текстур шляхом укладання копій цього зображення на зразок «черепиці», тобто розміщуючи копії текстури пліч-о-пліч так, щоб місця стику були непомітні. Тайли можуть бути прямокутні та ізометричні.

**API - Прикладни́й програ́мний інтерфе́йс,** набір визначень взаємодії різнотипного програмного забезпечення

d y = d y d x d x , {\displaystyle \mathrm {d} y={\frac {\mathrm {d} y}{\mathrm {d} x}}\,\mathrm {d} x,}

**ВСТУП**

Розвиток інформаційних технологій останні 25 - 30 років тісно пов'язаний з розвитком ігрової індустрії, а за останні 5-6 років зростає інтерес до розробки віртуальної реальності. А з цим зростає актуальність розробки фізичних рушіїв – програм, які моделюють фізичні закони реального світу у віртуальному.

Всі фізичні рушії діляться на ігрові та наукові. Наукові використовуються в науково-дослідних розрахунках і симуляції і володіють найбільшою точністю обчислень, оскільки, часто для них швидкість обробки інформації грає другорядну роль. Ігрові ж рушії схожі з науковими, проте спектр їх завдань менш широкий і на них покладаються досить прості завдання, для того, щоб в реальному часі прораховувати всі фізичні процеси, що відбуваються на сцені без шкоди швидкості роботи програми. В ігрових фізичних рушіях переслідується візуальна реалістичність симуляції фізичних процесів, для чого використовуються дуже сильні апроксимації, наближені моделі і інші програмні трюки.

Залежно від "масштабів" і специфіки гри, фізичний рушій, включений в них, може бути досить серйозним інструментом для оперативного розрахунку взаємодій безлічі тіл, які мають складну форму, неоднорідну масу щільність, енергію та інші атрибутами, або ж атрибути може геть-чисто відсутні.

Використання фізичного рушія може зробити ваш проект більш мобільним. Добре спроектований фізичний рушій дає можливість запускати проекти використовуючи різні платформи.

В роботі розглянуто алгоритми і метотоди розробики фізічних рушіїв твердих тіл, та їх реалізації.

**РОЗДІЛ 1. Фізичні рушії**

**1.1 Історія виникнення**

Сучасні фізичні рушії симулюють не всі фізичні закони реального світу, а лише деякі, причому з плином часу і прогресу в області інформаційних технологій і обчислювальної техніки список «підтримуваних» законів збільшується. На початок 2010 року фізичні рушії можуть симулювати такі фізичні явища і стану:  
динаміка абсолютно твердого тіла  
 -динаміка тіла, що деформується  
 -динаміка рідин  
 -динаміка газів  
 -поведінку тканин  
 -поведінку мотузок (троси, канати і т.д.)  
  
Розробники комп'ютерних ігор завжди ставили перед собою завдання поліпшення якості картинки, в той час як реалізація фізичних ефектів залишалася на другому плані. Звичайно, навіть в 90-і роки випускалися гри (серед них легендарна Half-Life), в яких присутня реалістична фізика, нехай і в малій кількості. З тих пір ситуація значно покращилася: розроблялися і випускалися фізичні рушії, про які сьогодні і піде мова.  
  
Фізичний рушій є програмою (або підпрограмою), яка відповідає за імітацію фізичних законів у віртуальному просторі. Рушії застосовуються як в ігрових додатках, так і в наукових. Основна відмінність ігрових рішень від наукових полягає в тому, що відтворена фізика не повністю відповідає дійсності, тобто існують деякі похибки. Багато в чому це обумовлено необхідністю швидкого прорахунку фізики в реальному часі. У випадку ж з науковими дослідженнями, перш за все, необхідна точність, а швидкість роботи не так важлива. Варто зазначити, що ігрові рушії використовують деякі спрощення для прискорення розрахунків. Так, простір може розбиватися на кілька складових для зменшення кількості перевірок зіткнень об'єктів. Цікаво, що при швидкому русі об'єктів їх зіткнення можуть не фіксуватися. У такому випадку використовуються технології безперервного відстеження зіткнень. На жаль, на даний момент фізичні рушії не можуть реалізувати всі фізичні закони. Вони працюють з динамікою абсолютно твердих і деформівних тіл, рідин, газів, а також тканин і мотузок.

Найбільш поширеними фізичними рушіями є NVIDIA PhysX, Havok, Open Dynamics Engine і Bullet Physics Library.

Найперший фізичний рушій був розроблений ірландською компанією Havok в 2000 році. У тому ж 2000 року Havok придбала німецьку компанію Ipion і її рушій Ipion Virtual Physics, тому вже наступні версії продукту Havok містили напрацювання Ipion. Серед них варто відзначити технологію відстеження зіткнень і систему зчленувань. Остання дозволяла розробникам відтворювати кулькове, пружинне та інші сполуки. На сьогоднішній день випущено сім поколінь Havok. Остання версія - Havok 7.1 - побачила світ 22 грудня 2009 року. Тут варто згадати про ще одну розробку компанії - рушія Havok FX. Його головним завданням було забезпечення прискорення «фізики» силами відеокарт AMD і NVIDIA, що дозволило б йому стати другим рушієм після Ageia PhysX з апаратною підтримкою, але після покупки Havok компанією Intel розробка проекту була згорнута. Проте, в 2008 році була досягнута домовленість між AMD і Havok про співпрацю і оптимізації фізичного рушія для продуктів американської компанії. Ще однією цікавою розробкою є технологія Havok AI, представлена ​​в 2009 році на конференції wскладу рушія Havok. Зараз можна нарахувати більше 250 ігор, які використовують фізичну складову Havok. Серед них Assassin's Creed, Fallout 3, Left 4 Dead, StarCraft 2 і багато інших.

**1.2 Застосування фізичних рушіїв**

Фізичні рушії дозволяють симуляцію подій реального світу використовуючи програмні системи, тому широко використовуються в областях, де потрібно чітко змоделювати певні події.

**Розробка ігор.** З появою все більшої кількості різних платформ, стала потреба пеерносити ігри з однієї платформи на іншу, при цьому затрачуючи на це якомога менше ресурсів, тому в більшості випадків під час розробки використовують фізичний рушій, щоб не було потреби писати багато зайвого коду та бібліотек.

**Віртуальна реальність.** Як форма комп’ютерного моделювання, що дозволяє користувачам якби зануритись у штучне середовище та діяти у ньому за домогою пристроїв, які пов’язують його реальні відчуття з комп’ютерною імітацією, якщо використовувати ці технології для, наприклад медичних цілей, коли використовуючи віртуальну реальність лікарі можуть проводити показові операції і т.д треба якісно і чітко моделювати фізику середовища, щоб віртуальний світ максимально передавав всі тонкощі справжнього.

**Верифікація фізичних систем.** Фізичні рушії вирішують проблему взаємодії фізичних тіл. Задавши математичну модель поведінки середовища, можна створити фізичну модель, якак буде максимально близька до реальної. За допомогою фізичних рушіїв, можна моделювати поведінку об’єктів на віддалених від землі астероїдах, планетах зірках чи кометах та проводити різні експерименти у віртуальному середовищі, при цьому заощаджуючи ресурси, які б треба було використати у реальному.

**Використання у медиціні.** У сучасній медицині активно використовують віртуальну реальність для лікування психічних розладів, таки як клаустрофобія, страх висоти чи посттравматичного синдрому, пацієнту симулють ситуацію від якої в нього виникає почуття страху, поступово змінюючи рівень детеалізації об’єктів.

**1.3 Типи реалізації поведінки об’єктів у 2D платформерах**

Можна виділити 4 типи вирішення проблеми реалізації поведінки персонажй у 2д платформері.

1. **Тайловий тип ( чистий)**

У цьому типі позиціонування об’єкту обмежено сіткою тайлів, таким чином, що він ніколи не зможе знаходитись між двома тайлами. Для створення ілюзії плавного пересування можуть бути використані різні анімації, однак, згідно з ігрової логіки, персонаж завжди перебуває прямо на конкретному тайлі. Це найпростіший спосіб зробити платформер, проте він накладає великі обмеження на контроль персонажа, роблячи його неприйнятним для традиційних екшн-платформер. Тим не менш, він дуже популярний для пазлів і «кінематографічних» платформерів.

Як це працює, на прикладі гри. Карта - це сітка тайлів, кожен з яких містить інформацію про властивості тайла: перешкода це чи ні, яке зображення використовувати, які звуки програвати і так далі. Рухомі та важливі об’єкти представлені набором з одного або декількох тайлів, що рухаються разом. Наприклад, в Lode Runner гравець є одним одним тайлом. У Toki Tori - гравець 2х2 тайла. А в грі Flashback, яка досить незвичайна через невеликого розміру сітки тайлів, гравець має габарити в два тайла в ширину і п'ять у висоту.

В іграх такого типу, персонаж зазвичай не рухається по діагоналі, а якщо і рухається, то, рух можна розкласти на два окремих кроки. Рух на кілька тайлів можна зробити множинними зрушеннями одиночних тайлів, якщо потрібно (у Flashback гравець завжди рухається по два тайла за один раз). Складається наступний алгоритм:  
  
   - Створити копію персонажа там, де він повинен виявитися (тобто якщо потрібно зрушити на 1 тайл вправо, потрібно зробити копію, де кожен тайл персонажа зрушать на один тайл вправо)

     -Перевірити цю копію на перетин з фоном та іншими персонажами  
    Якщо знайдено те, то пересування персонажа заблоковано. Потрібно відреагувати відповідним чином.

    - В іншому випадку шлях чистий. Наведіть персонажа сюди, відтворюючи анімацію, якщо необхідно, щоб переміщення виглядало плавним.

Недоліком такого типу руху є те, що він абсолютно непридатний для звичайних стрибків «по дузі», тому гри такого жанру зовсім позбавлені стрибків (Toki Tori, Lode Runner) або дозволяють тільки горизонтальні або тільки вертикальні стрибки фіксованої довжини (Prince of Persia, Flashback), такі стрибки є ніщо інше як звичайний лінійний рух. Переваги цієї системи - простота і точність. Такі ігри більш детерміновані, що веде до меншої ймовірності появи заторможень та більш чітко контрольованого геймплею, не вимагає занадто часто підлаштовувати значення в залежності від обставин. Фіксовані відстані для взаємодії дають можливість зробити красиву безшовну анімацію. Значно спрощується реалізація деяких ігрових механік (таких як хватка за виступи та односторонні платформи) - все що потрібно зробити, це перевірити, чи задовольняють тайли фону в потрібній позиції необхідним умовам.

Звичайно, ця система не дає робити кроки менш ніж на один тайл, але кроки можна зменшувати різними способами. Наприклад, тайли можуть бути трохи менше, ніж гравець (скажімо, гравець 2х6 тайлів), або можна дозволити «тільки візуальне» рух, щоб переміщатися всередині обраного тайла без зміни логіки (я вважаю, що це рішення застосоване в «Lode Runner - The Legend Returns »)є

**2. Тайловий тип (Плавний)**

У цьому типі Зіткнення як і першому визначаються сіткою тайлів, але об’єкти можуть рухатися у середовищі вільно (зазвичай з роздільною здатністю в 1 піксель.)

Такий тип часто використовували під час реалізації проектів для 8-ми і 16-ти бітних консолях, тип залишається популярним і сьогодні, так як вкрай легкий в реалізації і дозволяє редагувати рівень набагато простіше ніж з застосуванням більш складних технік. Також така реалізація дозволяє робити симуляцію ухилень та плавні стрибки по дузі.

Інформація про середовище зберігається так само я к і в чистому тайловому типі. Різниця полягає тільки у тому, як об’єкт взаємодіє з фоном. Тепер об’єкт описує деякий прямокутник, що використовується для прорахунку зіткнень (AABB, який не може обертатися), і, звичайно, за розміром є кратним розміром тайла. Стандартні розміри таких прямокутніків є один тайл в ширину і один (маленький Mario, пригнувшись Samus), два (великий Mario, Mega Man, пригнувшись Samus) або три (що стоїть Samus) тайла в висоту. У багатьох випадках візуально спрайт персонажа більше, ніж логічний прямокутник, так як це робить зовнішній вигляд гри більш приємним і геймплей - чеснішим (погодьтеся, що для гравця краще уникнути попадання, коли він повинен його отримати, ніж отримати коли не повинен).  
На зображенні вище, можна помітити, що спрайт з персонажем «X» квадратний (два тайла шириною), однак описує його прямокутник шириною не менше в один тайл.  
За умови, що немає ухилів і односторонніх платформ, алгоритм простий:  
  
     - Розкласти рух на осі X і Y, робити одне переміщення за раз. Якщо планується пізніше додати ухили, тоді спочатку по X, потім по Y. В іншому випадку порядок абсолютно не важливий. Потім для кожної осі:  
    Отримати координату межі в напрямку руху. Наприклад: якщо рухатися вліво, X координата лівій грані описує прямокутника. Якщо вправо, X координата правого боку. Якщо верх, Y координата верху і так далі.

   -  Визначити які лінії тайлів перетинаються з описуючим прямокутником - це дасть мінімальне і максимальне значення тайла на ІНШИЙ осі. Наприклад, якщо ми рухаємося вліво, припустимо гравець перетинається з горизонтальними лініями 32, 33 і 34

(ось воно, тайли з Y = 32 \* TS, Y = 33 \* TS і Y = 34 \* TS, де TS = розмір тайла).

    - Дослідити ці лінії з тайлами в напрямку руху, поки не знайдеться найближча перешкоду. Потім в циклі продивитсь на кожну рухомеуперешкоду, та визначити, яка з усіх перешкод найбільш близька на вашому шляху.  
    - Результуюче рух об’єкта вздовж цього напрямку -- це мінімум між відстанню до найближчої перешкоди і дальністю ходу нашого об’єкта.  
    Пересунути об’єкт на нову позицію. З цієї позиції обробляється інша координата, якщо ще не обробили.

1. **Бітова маска**

Метод бітової маски подібен до плавного тайлового методу, але замість використання великих тайлів використовується картинка для перевірки зіткнень для кожного пікселя. Це дозволяє краще пропрацювати гру, але і значно збільшує складність, використовує більше пам'яті і вимагає щось схоже з графічним редактором для створення рівнів. Така маска зазвичай не використовується безпосередньо для візуалізації, тому потрібні додаткові кошти - наприклад, велике графічне зображення (підкладка), індивідуально для кожного рівня. Через подібних проблем ця техніка досить рідкісна у використанні, але дозволяє домогтися більш якісних результатів, ніж варіанти, засновані на тайла. Цей метод зручний для створення динамічного оточення - руйнування можна просто «малювати» в бітову маску для зміни рівня.

Основна ідея дуже близька до тайловому (плавному) алгоритму - ми просто вирішуємо, що кожен піксель це тайл, і реалізуємо абсолютно той же алгоритм. Все буде працювати за одним невеликим винятком - ухили. Так як ухили тепер повністю визначаються относительностью позиції двох найближчих тайлів, попередня техніка не буде працювати і доведеться використовувати набагато складніший алгоритм замість неї.

Грубий опис алгоритму зміни позиції об’єкта:

    - Поєднуємо прискорення і напрям руху, щоб вирахувати вектор зміни позиції (наскільки рухатися по кожній осі)

    - Обробляємо кожну вісь окремо, починаючи з тієї, у якій абсолютна різниця більше.

   -  Для горизонтального руху зрушуємо AABB прямокутник гравця на 3 пікселі вгору, щоб він міг забиратися на ухили.

    -Сканувати далі, перевіряючи всі перепони і саму бітову маску, щоб обределіть на скільки пікселів можна зрушити до зіткнення з перешкодою. Переміщаємо на нову позицію.

    -Якщо це було горизонтальне рух, рухаємо на стільки пікселів вгору, наскільки необхідно (взагалі воно повинно бути не більше трьох) щоб забратися на ухил.

    - Якщо в кінці пересування будь-піксель персонажа перетинається з будь-якою перешкодою, прибираємо пересування по цій осі.

    - Згідно результату останньої умови, робимо те ж саме для іншої осі.

1. **Векторний тип**

Ця техніка використовує векторні дані (лінії або багатокутники) для визначення граней зони зіткнень. Незважаючи на дуже велику складність в коректній розробці, вона стає все більш поширеною завдяки великій кількості фізичних движків, таких як Box2D, які підходять для реалізації цієї техніки. Вона дає всі принади техніки бітових масок, але без величезної перевантаження пам'яті і використовує зовсім інший метод редагування рівнів.

Ось два основні підходи до реалізації:  
  
     Обробляти рух і зіткнення самостійно, схоже з бітовими масками, але використовуючи багатокутники для прорахунку перетинів об'єктів і руху.  
     Або ж реалізувати бібліотеку, в якій будуть описані основні фізичні функці, іншими словами, як раз для такого типу симуляцій у 2D і потрібно використовувати фізичні рушії

**РОЗДІЛ 2. Стадії роботи фізичного рушія**

У роботі більшу увагу надано фізичному рушію твердих тіл(rigid body), такий рушій повинен вміти виконувати запити виду «додати тіло», «симулювати крок часу», «взяти позицію / орієнтацію даного тіла» та інші. Розглянемо докладніше основну його функцію «симулювати крок часу», що виконується кожен кадр в реалтайм додатку, наприклад, в грі. У кожному кадрі ми перш за все обов'язково повинні знайти всі потенційно взаємодіючі пари тел. Цей крок називається Broadphase, це є перший крок для визаченя колізій.

**2.1 Визначня колізій**

Алгоритми виявлення зіткненнь(колізій) в 2D-випадку залежать від типу фігур, які можуть стикатися (наприклад, Rectangle to Rectangle, Rectangle to Circle, Circle to Circle). Як правило, у вас буде проста загальна форма, яка охоплює сутність, відому як «hitbox», тому, хоча зіткнення не може бути досконалим на піксель, воно буде виглядати досить добре і працювати на декількох об'єктах. Даліі представлений огляд найбільш поширених методів, використовуваних для забезпечення виявлення колізій у 2D середовищі.

**Axis-Aligned Bounding Box.** Однією з простих форм виявлення зіткнень між двома прямокутниками, які по осі вирівняні - означає відсутність обертання. Алгоритм працює шляхом забезпечення немає зазору між будь-який з 4-х сторін прямокутників. Будь розрив означає, що зіткнення не існує.

Приклад реалізації:

var rect1 = {x: 5, y: 5, width: 50, height: 50}

var rect2 = {x: 20, y: 10, width: 10, height: 10}

if (rect1.x < rect2.x + rect2.width &&

rect1.x + rect1.width > rect2.x &&

rect1.y < rect2.y + rect2.height &&

rect1.height + rect1.y > rect2.y) {

// collision detected!

}

// filling in the values =>

if (5 < 30 &&

55 > 20 &&

5 < 20 &&

55 > 10) {

// collision detected!

}

**Зіткнення кіл Circle Collision**  
  
Ще одна проста форма для виявлення зіткнень між двома колами. Цей алгоритм працює з центральними точками двох кіл і забезпечити відстань між центральними точками менше двох радіусів підсумовуються.

var circle1 = {radius: 20, x: 5, y: 5};

var circle2 = {radius: 12, x: 10, y: 5};

var dx = circle1.x - circle2.x;

var dy = circle1.y - circle2.y;

var distance = Math.sqrt(dx \* dx + dy \* dy);

if (distance < circle1.radius + circle2.radius) {

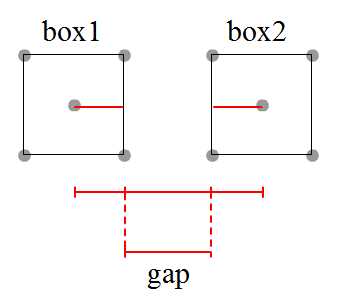
// collision detected!

}

**Теорема розділової осі SAT theorem**

SAT- теорема часто використовується для перевірки зіткнень між двома простими багатокутниками, або між полігоном і окружністю. Як і з усіма алгоритмами, вона має свої сильні і свої слабкі сторони. SAT по суті стверджує, якщо ви взмозі провести лінію для поділу двох багатокутників, то вони не створюють колізію.

Давайте припустимо тепер, що багатокутники ми називаємо квадрати: box1 зліва і Box2 справа. Легко бачити, що ці квадрати по горизонталі розділені. Прямолінійний підхід, щоб визначити це в коді, щоб обчислити горизонтальне відстань між двома квадратами, а потім відняти півширини

рис 1.

/псевдокод для оцінки поділу of box1 та box2 такий:

var length:Number = box2.x - box1.x;

var half\_width\_box1:Number = box1.width\*0.5;

var half\_width\_box2:Number = box2.width\*0.5;

var gap\_between\_boxes:Number = length - half\_width\_box1 - half\_width\_box2;

if(gap\_between\_boxes > 0) trace("It's a big gap between boxes")

else if(gap\_between\_boxes == 0) trace("Boxes are touching each other")

else if(gap\_between\_boxes < 0) trace("Boxes are penetrating each other")

**2.2 Представлення колізій:**

Більшість алгоритмів для виявлення зіткнень досить просто обчислити, це може бути марною тратою циклів для перевірки \* кожен \* об'єкт з будь-яким іншим об'єктом. Зазвичай гри будуть розділені зіткнення на дві фази, широку та вузьку.

**1.Broadphase** У кожному кадрі ми перш за все обов'язково повинні знайти всі потенційно взаємодіючі пари тел. Цей крок називається Broadphase. Щоб звузити кількість пар для перевірок, застосовуються алгоритми:  
  
**Sweep And Prune**

Ідея визначення перетину AABB vs AABB заснована на сортуванні і оновленні інтервалів в real time. Нехай задано n інтервалів Ii = [bi, ei], де 1 <= i <= n. Завдання полягає в тому, що б ефективно знайти всі пересічні пари інтервалів. Ii і Ij перетинаються тільки в тому випадку, якщо bj <= ei і bi <= ej. Саме просто рішення для перевірки всіх пар інтервалів має складність O (n \* n). Більш ефективний підхід використовує sweep алгоритм:

  1. Сортуємо все кінцеві точки інтервалу по зростанню.

  2. Для всіх інтервалів в залежності від типу точки оновлюємо (якщо початкова точка інтервалу - додаємо, інакше видаляємо) безліч A, яке містить пересічні інтервали.

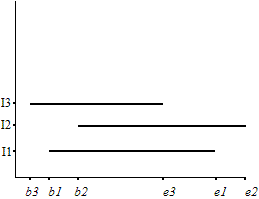


Рис.2

1. b3, немає ніяких перетинів, оскільки A пусте. Оновлюємо A = {I3}.  
   2. b1, перетин I3 і I1. Оновлюємо A = {I3, I1}.  
   3. b2, перетин I3 і I2, і I1 і I2. Оновлюємо A = {I3, I1, I2}.  
   4. e3, видаляємо I3 з безлічі A. A = {I1, I2}  
   5. e1, видаляємо I1 з безлічі A. A = {I2}  
   6. e2, видаляємо I1 з безлічі A. A = {}

В 2d, наприклад, для прямокутників необхідно перевірити те інтервалів по двох осях.  
  
Розглянемо в 3d перетин AABB vs AABB.  
  
AABB заданий як min, max.  
  
Для кожного проходу алгоритму вибирається вісь, уздовж якої буде проходити сортування. Знаючи вісь, покажчики на AABB упорядковано в зростаючому порядку по min значенням. Після того як масив відсортований, перевіряємо на непересічні AABB, тобто перевіряємо AABBj.min [SORT\_AXIS]> AABBi.max [SORT\_AXIS], якщо істина, то інші n-j перевіряти не треба.  
  
В такому випадку можна досягти складності в O (n log n) або в O (kn), в залежності від сортування.  
  
Як вибирати вісь, уздовж якої сортувати. Наприклад, можна порахувати дисперсію (min + max) / 2 AABB. І в залежності від значення вибирати потрібну вісь. Але навіть в такому випадку можливий випадок в O (n \* n).

Реалізація методу:

int SORT\_AXIS = 0;

struct AABB

{

Point min;

Point max;

};

int cmpAABB(const void \* a, const void \* b)

{

float mina = (\*(AABB\*\*)a)->min[SORT\_AXIS];

float minb = (\*(AABB\*\*)b)->min[SORT\_AXIS];

if(mina < minb) return -1;

if(mina >= minb) return 1;

return 0;

}

AABB \* vAABB[N];

void SweepAndPrune()

{

//Перша частина алгоритму: сортування по осі SORT AXIS

qsort(vAABB, N, sizeof(vAABB\*), cmpAABB);

// Суми для підрахунку середнього квадратичного.

Point s (0,0,0);

Point s2 (0,0,0);

//шукаємо інтервали що перетинаються.

for (int i = 0; i < N; ++i)

{

AABB \* aabb = vAABB[i];

/\*

Дисперсію будемо вважати відносно середнього min, max.  
     Можна вибрати будь - то іншу характеристику для підрахунку.

\*/

Point p = (aabb->min + aabb->max) \* 0.5;

s+=p;

s2+=p\*p;

for (int j = i + 1; j < N; ++j)

{

if (vAABB[j]->min[SORT\_AXIS] > aabb->max[SORT\_AXIS] )

break;

if(AABBOverlap(vAABB[j], aabb))

{

/\*

Перевіряємо колізії складних об'єктів, або якісь інші корисні дії \*/

}

}

}

// Рахуємо вибіркову дисперсію, і в залежності від значення вибираємо вісь.

v = s2 / N – pow(s / N, 2);

SORT\_AXIS = 0;

if ( v[1] > v[0] ) SORT\_AXIS = 1;

if ( v[2] > v[SORT\_AXIS] ) SORT\_AXIS = 2;

**QuadTree / Octree** і їх розширення, такі, як loose tree - алгоріми по розподіленню простору (Spacial Hashing). Ґрунтуються на звуження кола пошуку контактуючих тіл в рамках звуженої області простору, наприклад, немає сенсу намагатися шукати контакт чайника, що стоїть на столі з автомобілем, що стоїть під вікном.

**Temporal coherence** Багато алгоритмів, що грунтуються на так званому ефекті Temporal coherence, який можна тлумачити як «тимчасова когерентність». Суть полягає в тому, що тіла від кадру до кадру зазвичай зсуваються на малу відстань, і тіла, які на даній ітерації знаходяться далеко один від одного, на наступній напевно будуть знаходитися трохи ближче. Тобто принцип тимчасової когерентності полягає у тому, що за малий крок часу стан системи зазвичай змінюється незначно. На цій властивості засновані більшість алгоритмів інтеграції позиції, Солвер та більшість алгоритміввизначення зіткнень.

**Narrowphase** Далі, знайшовши пари потенційно пересічні тіла, вступає в силу Narrowphase - стадія, на якій движок повинні знаходити точні дані про контакти тіл - знаходити точки контакту, нормалі та і глибини проникнень.

Під час цієї стадії визначається зіткнення для пари потенційно пересічних геометрій, знайдених в широкій фазі (Broadphase). Зазвичай є найбільш трудомісткою операцією, так як саме в вузькій фазі алгоритму потрібно знайти всі точки контакту з нормалями і глибинами проникнення.  
  
У фізиці твердих тіл, при пошуку колізій конвексів(convex) варто відзначити основні алгоритми:  
  
Алгоритм, безпосередньо використовуючий SAT- одну з основних теорем, на якій побудовані більшість інших алгоритмів, цю теорему ми вже розглядали в цьому розділі  
  
V-Кліп / Лін Канни - алгоритми, засновані на властивостях Областей Воронова ((Voronoi Features). Найкраще підходить для моделювання «непроникаючої фізики» -, алгоритми забороняють будь-які взаємопроникнення тіл.  
  
GJK, EPA - одні з найпотужніших алгоритмів, що базуються на математичному об'єкті «Minkovski Sum», або Сумі Міньківського. На відміну від багатьох інших, ці два можуть працювати не тільки з многогранниками, але і з будь-якими опуклими тілами.

**Solver** Солвер (Solver) фізичного рушія - одна з основних систем рушія, що відповідає за дозвіл контактів та інші види зв'язків. У свою чергу теж поділяється на дві підзадачі:  
  
     - вирішення рівнянь для швидкостей, щоб дві кулі, зіткнувшись, розлетілися в потрібних напрямках;

     - вирішення рівнянь для позицій, щоб та ж два куля, «випадково» проникнувши один в одному, успішно «виліз» і щоб це ще по можливості смотерлісь досить природно.

Під час наступної стадії solver займається розрахунком та дозволами знайденихх контактів та інших типів зв'язків.

Завершальна стадія - інтегрування позиція. Тобто, власне, зміщення всіх тел сцени. Стадія не надто складні, вибір алгоритми її реалізації невеликі, але зате вони відмінно працює - інтегратори по Ейлера, Ньютон і Рунг-Кутта.

**РОЗДІЛ 3. Порівнняня існуючих фізичних рушіїв**

**Godot :**

**Жанрова спрямованість:** 2D / 3D гри будь-якого жанру і типу;  
**Платформи конструктора:** Mac OS X 32, Windows 32/64, Linux 32/64, Linux Server 64;  
**Платформи для експорту ігор:** Windows 32/64, Linux 43/64, Mac OSX, Android, HTML5 (список обіцяє поповнитися в тому числі консолями і flash-плеєром);  
**Ліцензія:** вільна (MIT);  
**Відкритий вихідний код:** вільно надається;  
**Графічний API:** OpenGL ES 2;  
**Мова програмування:** вбудована скриптова мова GDScript;  
**Переваги: ​​**один з небагатьох конструкторів нативної працює під Linux;  
**Недоліки:** не всі заявлені можливості реалізовані;  
**Розробники:** Juan Linietsky і Ariel Manzur з Okam Studio;  
**Початок розробки:** 2001 рік;  
**Перший реліз:** лютий 2014 року;  
**Реліз першої стабільної версії:** 16 грудень 2014 року.  
  
Godot - це безкоштовний, відкритий, кросплатформенних движок і конструктор для створення ігор від студії Okam.  
  
Процес написання ігор за допомогою конструктора Godot складається з створення ієрархії об'єктів різних типів, що відповідають тому чи іншому аспекту ігрової механіки, і написання поведінки для них на вбудованому скриптовій мовою GDScript.  
  
Ієрархії цих об'єктів об'єднуються в сцени. Сценами можна не тільки розділяти ігровий процес, відображаючи їх по одній. Так само будь-яка сцена може бути додана в ієрархію елементів будь-якої іншої сцени. Таким чином можна вже написану 2D сцену гри в пінг-понг завантажити в якості текстури для полігону інший, вже 3D сцени. Або просто в сцени можуть бути винесені повторювані елементи, для зручного повторного використання.

**Box2D**

**Жанрова спрямованість**: 2D гри будь-яких жанрів і типів;  
**Платформи:** PC Windows, Flash, iOS (iPhone, iPad), Java;  
**Мова програмування:** C ++, Java, Java Script, AS3, C #, Python, Delphi;  
**Ліцензія:** безкоштовна, ZLIB;  
**Відкритий вихідний код:** вільно надається;  
**Переваги: ​​**висока продуктивність, багатоплатформовий, відкритий і безкоштовний;  
**Недоліки**: рідко оновлюється;  
**Автори движка:** для Windows Ерін Катто (Erin Catto) та інші.  
  
Box2D - це відкритий двомірний фізичний движок, що розробляється на Visual C ++ під Windows програмістом Еріном Катто (Erin Catto). Даний движок забезпечує обробку плоскої фізики і колізій, залишаючи розробникам вибирати графічний рендер на свій розсуд.  
  
Спочатку це бібліотека для моделювання реалістичною двомірної фізики на C ++. Однак існують версії і для інших мов програмування, таких як Java, Java Script, Action Script 3 (Flash), Python, C #, Delphi, DarkBASIC від The ​​Game Creators Ltd. та ін.  
  
У Box2D можна встановлювати обмеження на об'єкти: двомірне тіло має 3 ступеня свободи (обертання і пересування по горизонталі і вертикалі), будь-яку з яких можна заблокувати. Також можна поєднувати різні об'єкти, всього є п'ять типів з'єднань:  
  
Жорсткий відрізок - центри тіл, з'єднаних таким чином, знаходяться весь час на однаковій відстані одна від одної.  
Болтове - тіла обертаються відносно один одного навколо точки їх з'єднання. Прикладом такого з'єднання тел може служити суглоб людини.  
Призматичне - дозволяє тіл переміщатися лише щодо певної осі (заданої щодо тел) і не допускає обертання цих тіл.  
Талеві - талі з'єднують два тіла з землею і один з одним. Загальна довжина двох сегментів талей фіксована.  
Передавальне - одна ступінь свободи впливає на будь-яку іншу ступінь свободи іншого тіла.  
  
Ще Box2D підтримує фільтри зіткнень. Це коли всі об'єкти розподіляються по різних групах, і між групами встановлюються відносини - чи будуть стикатися об'єкти цих груп між собою або просто пролітати крізь один одного.  
  
API бібліотеки не дуже складний, досвідченому програмісту не складе труднощів в ньому розібратися. Тим більше, що для Box2D є посібник російською мовою.  
  
Цей фізичний движок використовувався в безлічі різних 2D-ігор. Наприклад, він використовувався студією розробників ігор Kloonigames в їхньому проекті "Crayon Physics Deluxe", який був відзначений головним призом на фестивалі IGF 2008.

**Cocos2d**

**Жанрова спрямованість:** 2D гри будь-якого типу  
Платформа: iOS, Android, Windows Phone 8, Windows, Linux, Tizen, Mac OS X, XBox360, HTML5;  
**Ліцензія**: MIT License;  
**Мови програмування:** C ++, lua, Javascript, C #, Objective-C, Python;  
Відкритий вихідний код: повністю відкритий;  
**Графічний API:** OpenGL;  
**Переваги:** ​​безкоштовний, швидкий, безліч за замовчуванням інтегрованих сторонніх бібліотек, постійні оновлення, мультиплатформеність;  
**Недоліки:** Практично неможливий для освоєння новачками;  
  
Сімейство двигунів бере початок в 2008 році (за інформацією з Вікіпедії) і згодом розростається на безліч різних бібліотек, що відрізняються мовами, орієнтованими платформами, швидкодією, вбудованими сторонніми бібліотеками та іншим.  
  
На даний момент існує 5 актуальних версій - Cocos2d-x, Cocos2d-JS (Є JS орієнтованої версією Cocos2d-x), Cocos2D-XNA, Cocos2D-Swift, і родоначальник Cocos2d, написаний на Python, а так само підтримується кілька редакторів (Конструкторів) :  
  
CocoStudio: інструментарій на основі Cocos2d-x, містить UI Editor, Animation Editor, Scene Editor і Data Editor; разом утворюючи закінчену систему.  
CocosBuilder: безкоштовне ПО під ліцензією MIT.  
SpriteBuilder: SpriteBuilder - нове покоління CocosBuilder.  
Багато редакції, що характерно для open sourse-софта підтримують сторонні бібліотеки за замовчуванням, такі як, наприклад Box2D, Chipmunk, OpenAL.  
  
Хоча рушій орієнтований на 2D, з недавніх пір Cocos2D-x підтримує роботу з 3D (А так же ізометричне надання графіки), що, безсумнівно, є великим плюсом.

**Coherent UI**

 **Жанрова спрямованість:** інтеграція мережевих елементів з 2D / 3D-іграми;

 **Платформи:** Windows, Linux, Mac OS X, Google NaCl, .NET, iOS, Android, Xbox 360, PlayStation 3, Unity3D, CryEngine і багато інших

 **Графічний API:** DirectX і OpenGL;

 **Ліцензія:** комерційна, є безкоштовна версія;

 **Мови програмування:** C ++, а також HTML, CSS і JavaScript;

 **Відкритий вихідний код:** надається ліцензіатам;

 **Переваги:** спрощення інтеграції, наявність шаблонів і ком'юніті;

 **Недоліки:** відносно висока ціна;

 **Розробники:** Coherent Labs.

Coherent UI - це призначений для користувача інтерфейс для ігор. Ця система розроблена спеціально для ігор, щоб можна було використовувати елементи на стандартних HTML, CSS і JavaScript для взаємодії безпосередньо в іграх, накладаючи елементи на будь-які, навіть тривимірні об'єкти.

З її допомогою можна без особливих зусиль реалізовувати ігрові елементи, інтегровані з мережею. Наприклад, прямо з гри можна переглядати різні сервіси з підключенням до мережі, типу Facebook і Youtube.

Основні можливості:

 Показ web-сайтів на будь-якому об'єкті;

 Підтримка HTML5 / CSS3 і SSL;

 Соціальна інтеграція з Facebook, Twitter і Google+;

 JavaScript для зв'язують взаємодій;

 Живе редагування та налагодження інтерфейсу;   
і т.д.

Також розробники надають різні доповнення, типу Browser Kit і Social Kit. І працюють над Chat Kit, FPS Kit, MMO Kit, Micro-transaction Kit і т.д.

**РОЗДІЛ 4. Розробка**

Для початку потрібно усвідомити відміну комп'ютерної фізики від реального. Реальна фізика діє безперервно (в усякому разі протилежне не довести на поточний момент). Комп'ютернафізика, як і комп'ютер діє дискретно, тобто ми не можемо обчислювати її безперервно, тому розбиваємо її обчислення на кроки з певним інтервалом (я віддаю перевагу інтервал 25 мс). Координати об'єктів змінюються після кожного кроку і об'єкти виводяться на екран.  
  
Тепер приступимо до самої гравітації.  
  
Закон всесвітнього тяжіння (Ньютонова гравітація) говорить:

F = G \* m1 \* m2 / R^2

де:  
  
F [Н] - сила тяжіння між двома об'єктами  
G = 6,67 \* 10 ^ -11 [м ^ 3 / (кг \* с ^ 2)] - гравітаційна постійна  
m1, m2 [кг] - маси 1 і 2 об'єктів  
R [м] - відстань між центрами мас об'єктів.

До об’єктів будемо застосовувати силу, використовуючи другий закон Ньютона – F=m\*a, де a [м/с^2] – прискорення об’єкту.

Нехай d –диференціал(похідна), v – швидкість, s – відстань, а Pos – точка(координати об’єкта у цей час).

переходимо від векторів до скалярів:

ax = a \* cos(α)

ay = a \* sin(α)

dv.x = ax \* dt

dv.y = ay \* dt

vx += dv.x

vy += dv.y

ds.x = vx \* dt

ds.y = vy \* dt

Pos.x += ds.x Pos.y += ds.y

де:

cos(α) = dx / R

sin(α) = dy / R

dx = Pos2.x - Pos.x dy = Pos2.y - Pos.y

R^2 = dx^2 + dy^2

Так як іншого виду сили в проекті поки немає, то використовуємо (1) в такому вигляді і трошки полегшимо обчислення:

F = G \* m \* m2 / R^2 a = G \* m2 / R^2

Тоді крок оновлення фрейму буде обчислюватись наступною функцією:

function Step(){

var a, ax, ay, dx, dy, r;

// важливо зробити розрахунок для кожного об’єкту

for(var i = 0; i < star.length; i++) // рахуєм теперишній

for(var j = 0; j < star.length; j++) // рахуєм другий

if(i == j) continue;

dx = star[j].x - star[i].x;

dy = star[j].y - star[i].y;

r = dx \* dx + dy \* dy;// тут R^2

if(r < 0.1) r = 0.1; // щоб не ділилось на дуже мале число

a = G \* star[j].m / r;

r = Math.sqrt(r); // тут R

ax = a \* dx / r; // a \* cos

ay = a \* dy / r; // a \* sin

star[i].vx += ax \* dt;

star[i].vy += ay \* dt;

}

for(var i = 0; i < star.length; i++){

star[i].x += star[i].vx \* dt;

star[i].y += star[i].vy \* dt;

}

// вивод результату на екран.

Draw();

}

Крім того, що складність такого алгоритму росте експоненційно, об’єкти пролітають один скрізь одний, тому треба ще забезпечити прорахунок колізій, а саме коліжон детекшон, який було описано в минулих розділах.

**Реалізація collision detection**

Додавши алгоритм знаходження колізій, тепер можна виявляти зіткнення між кілами(з використанням регіонів Вороного), та зіткнення між опуклими багатокутниками. Для кожного типу об’єкта створено свій клас, такий як клас для точки, кола чи коробки. Додано об'єкт, що являє собою результат зіткнення між двома об'єктами. У нього є тільки простий new Response() конструктор. Колізії різниїх об’єктів між собою представлені класами кола з колом, кола з полігоном, полігона з колом та точка з об’єктом.

Приклад реалізації зіткнення двох кіл:

var V = SAT.Vector;

var C = SAT.Circle;

var circle1 = new C(new V(0,0), 20);

var circle2 = new C(new V(30,0), 20);

var response = new SAT.Response();

var collided = SAT.testCircleCircle(circle1, circle2, response);

// collided => true

// response.overlap => 10

// response.overlapV => (10, 0)

Зіткнення двох полігонів:

var V = SAT.Vector;

var C = SAT.Circle;

var P = SAT.Polygon;

var circle = new C(new V(50,50), 20);

// A square

var polygon = new P(new V(0,0), [

new V(0,0), new V(40,0), new V(40,40), new V(0,40)

]);

var response = new SAT.Response();

var collided = SAT.testPolygonCircle(polygon, circle, response);

// collided => true

// response.overlap ~> 5.86

// response.overlapV ~> (4.14, 4.14) - i.e. on a diagonal

**ВИСНОВКИ**

В роботі розглянуто підходи до реалізаці біблотек для симуляції поведінки твердих тіл, а саме симуляції того, як тіла взаємодіють між собою, було роглянуто відомі фізичні рушії та порівняно їх, було розроблено реалізацію знаходження колізій між твердими тілами.

Під час запуску рушія, було встановлено, деякі недоліки, такі як – під час падіння кіл однаковою форми один на одний, вони не піддавались силі тяжіння, та вели себе так як і чотирикутники.

Встановлено, що для написання невеликих симуляцій поведінки тіл, краще створити власний рушій, а не тягнути бібліотеки вже відомих, бо в цих бібліотеках є забагато зайвих функцій, що не використовуються, а тому такі бібліотеки тільки ускладнять проект.

**СПИСОК ЛІТЕРАТУРИ ТА ПЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ**

1. Physics, Part 3: Сollision Response , Chris Hecker
2. Gcup, архів бібліотек для розробки ігор [https://gcup.ru/load/libraries/]
3. Mozilla Developer Network Techniques for game development

[https://developer.mozilla.org/en-S/docs/Games/Techniques/2D\_collision\_detection]

4. «Физический движок: взгляд изнутри [https://tproger.ru/translations/whats-in-a-projectile-physics-engine-part-3/]

5. список терминов и понятий, относящихся к программированию физических движков [http://www.gamedev.ru/terms/?div=code&subdiv=physics]

# 6. Rigid body object type [https://wiki.blender.org/index.php/Doc:2.4/Manual/Game\_Engine/Physics/Object\_Type/Rigid\_Body]

# 7. Projectile Physics Engines: Building a Game World

[https://gamedevelopment.tutsplus.com/tutorials/projectile-physics-engines-building-a-game-world--cms-21811]