|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  |  | |
|  | | | | |
|  | | для прик эмбл |  | |
| МИНОБРНАУКИ РОССИИ | | | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **"Московский государственный технический университет радиотехники,**  **электроники и автоматики"**  **МГТУ** **МИРЭА** | | | | |
| Факультет Кибернетики  *(наименование факультета)* | | | | |
| Кафедра “Вега”  *(наименование кафедры)* | | | | |

|  |  |
| --- | --- |
| **КУРСОВОЙ ПРОЕКТ (РАБОТА)** | |
| **по дисциплине** | |
| **«**Языки и методы программирования**»**  *(наименование дисциплины)* | |
| **Тема курсового проекта (работы) «Игра Space Invaders»**  *(наименование темы)* | |
| Студент группы КМБО-02-15  *(учебная группа)* | *Пистун Е. Н.* |
| Руководитель курсового проекта (работы)  *должность, звание, ученая степень* | *Милонов Г. А.* |
| Рецензент (*при наличии*)  *должность, звание, ученая степень* | *\_\_\_\_\_\_\_\_\_\_\_\_* |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работа представлена к защите | «\_\_»\_\_\_\_\_\_\_2016 г. | *(подпись студента)* |
|  |  |  |
| «Допущен к защите» | «\_\_»\_\_\_\_\_\_\_2016 г. | *(подпись руководителя)* |

# 

Москва 2016 г.

Оглавление

[**Введение**](#_sbv7a0ualk0g) **3**

[**Описание задачи и её решения**](#_xbxktojjk0hv) **3**

[**Руководство пользователя**](#_oaav316v4sqc) **3**

[Запуск](#_ce1b6vxxokcx) 3

[Управление](#_9ep2zx16xbc8) 3

[Механика игры](#_2g8kn0pyumtv) 4

[**Документация**](#_frfxbek15hyj) **5**

[Сборка программы](#_be9gl0di3kyn) 5

[Debian/Ubuntu](#_d2m9x0fpx2br) 5

[Linux](#_n7aqrxo9nxvj) 5

[Движок](#_h6199e2lied9) 5

[Инициализация](#_ctaepf3ff37j) 5

[Главный цикл](#_o5s1b01p540) 5

[pse::GameState](#_ab95qb8kua7q) 6

[Графика](#_vc0ozmyo6u6m) 6

[pse::RenderInfo](#_2jniwcjalygp) 6

[pse::TextureManager](#_e882yyk0oezw) 7

[pse::ShaderManager](#_cmh3f6ph3xxb) 7

[pse::ResourceManager](#_wi1dia3ewnk5) 7

[pse::Texture](#_apdl0hk69y2r) 7

[pse::Shader](#_ax89u2bmfz9r) 8

[pse::Vertex](#_7a3gu6xw13i9) 8

[pse::Mesh](#_3qa38fgehkn2) 8

[pse::Framebuffer](#_w9ls11xqpli2) 9

[pse::Sprite](#_52ct8rbidevw) 9

[pse::RenderManager](#_ame8pkop1gbo) 10

[pse::WindowManager](#_eeg4pvbuv1ug) 10

[pse::Font](#_mlv4026u8w32) 10

[pse::Font::String](#_e8wz9of8algu) 11

[pse::Camera](#_if8d8ll3n69u) 11

[Таймеры](#_vqdewy8qwo9i) 11

[pse::Timer](#_jpgdhr7vf71h) 11

[Интерполяция](#_xdzt8horcmx1) 13

[pse::Interpolator](#_33femi9k9ud5) 13

[pse::InterpolatorQueue](#_vcvb9bn86fhr) 13

[Игра 0xff2](#_o2jbvvr8f3pt) 14

[Сущности, Entity](#_xc4h2o5l8iwk) 14

[Игрок, Player](#_qieqghm4258) 14

[Враг, Enemy](#_z1by5avymkny) 15

[Пули, BulletType](#_qp6utvra1hs1) 15

[Пули, BulletInstance](#_jlnzckerichd) 16

[Пули, BulletHell](#_9slvx3qi473) 16

[Хитбокс, Hitbox](#_lpnzqeioo70m) 16

[Состояние игры, StateInfo](#_su7pikxg6raw) 17

[Начальный экран, GameStateStartScreen](#_2kqjijoyqgck) 17

[Экран игры, GameStateGame](#_39kj9efg9u19) 17

[**Заключение**](#_1a6fanf48i5) **18**

[**Приложение**](#_xyov8s4607f2) **18**

[Скриншоты](#_cqg9cfgq1a6n) 18

# 

# 

# 

# Введение

Тема данной курсовой работы – создание игры в стиле “Space Invaders”. Необходимо было проанализировать механику игры, выбрать технические средства для её реализации.

Данная работа состоит из нескольких частей, включая описание задачи и решения, руководство пользователя, документация (документация игро

вого движка и непосредственно игры).

# Описание задачи и её решения

Механика создаваемой игры достаточно тривиальна – игрок управляет неким героем, который должен уничтожать вражеских героев и избегать потери своих “жизней” (попыток).

Графика игры полностью состоит из спрайтов. Управление происходит с помощью клавиатуры.

Для реализации управления через клавиатуру была выбрана библиотека SDL2, реализующая событийную модель для обработки данных с клавиатуры.

Для реализации графики было принято решение создать некий “игровой движок”, который обеспечивает удобную работу с двумерной графикой (с возможностью использования трехмерной графики). Для этого были выбраны следующие средства: SDL2 для инициализации окна и контекста OpenGL, SDL2 Image для загрузки изображений в качестве текстур, glew для управления контекстом OpenGL, glm для некоторых математических функций и операций с матрицами, OpenGL 3.1 и GLSL для отрисовки графики посредством шейдеров.

# 

# Руководство пользователя

## Запуск

Запустить исполняемый файл povis из дистрибутива программы (см. приложение).

## Управление

При запуске игры открывается заставка. Для того, чтобы перейти к игре, необходимо нажать кнопку Enter. Чтобы выйти из игры, необходимо нажать кнопку Esc.

После перехода к игре действует следующее управление:

* F - переключение между оконным и полноэкранным режимом
* Стрелки / WASD - движение персонажа по игровому пространству
* Пробел / Z - выстрел
* Долгое нажатие R - запустить уровень сначала
* Esc - возвращение на экран заставки

## Механика игры

Игрок управляет главным героем. Герой:

* Имеет 3 жизни
* Может стрелять (создавать пулю перед собой, движущуюся снизу вверх)
* При попадании в героя пули, созданной врагом/боссом, игрок теряет одну жизнь

На экране также существуют враги. Каждый враг:

* Имеют одну жизнь
* После попадания во врага пули, созданной главным героем, враг умирает и исчезает с игрового поля
* После смерти врага, скорость движения врагов несколько увеличивается, увеличивая сложность игры
* В случайном порядке создает пули, движущиеся сверху вниз
* Двигается в строю с другими врагами по горизонтали
* Строй врагов отталкивается от краев экрана

После уничтожения всех врагов, появляется босс. Босс:

* Имеет 20 жизней
* При попадании в босса пули игрока, счётчик жизней босса уменьшается на один
* Когда счётчик жизней достигает нуля, босс умирает и исчезает с экрана
* Босс создает пули по заданному шаблону

Также присутствует механика “очков”.

* Игрок начинает игру, имея 100 очков
* Каждую секунду игрок теряет 3 очка
* За каждого убитого врага игрок получает 10 очков
* За каждую потерянную жизнь игрок теряет 10 очков
* За убийство босса игрок получает 1000 очков

Игра заканчивается в двух случаях:

* Счетчик жизней игрока достигает нуля (поражение)
* Убит босс (победа)

В любой момент можно начать игру заново (долгое нажатие R).

# Документация

## Сборка программы

### Debian/Ubuntu

Для систем, основанных на Debian (например, Ubuntu), перед сборкой необходимо установить следующие библиотеки через *apt-get*:

*apt-get install libsdl2-dev libsdl2-image-dev libglm-dev libglew-dev*

Кроме того, необходим *cmake* и *make*.

После установки библиотек, можно собрать проект с игрой, выполнив в корневой папке исходного кода следующие команды:

*cmake .*

*make*

После завершения сборки программу можно запустить, просто вызвав исполняемый файл: *./povis*

### Linux

Сборка на других linux-системах также возможна, с точностью до того, что указанные библиотеки необходимо установить, используя инструкции конкретного дистрибутива.

## Движок

В данный момент движок реализует работу с двумерной графикой (на основе OpenGL 3.1 и GLSL, благодаря чему возможно использование и трёхмерной графики), таймерами (в тиках игрового цикла), интерполяцию и стек состояний игры.

### Инициализация

Главный класс движка – синглтон *pse::Game*. Для инициализации необходимо в главной функции программы вызвать:

*pse::Game::initialize(string title, unsigned width, unsigned height);*

Эта функция инициализирует синглтон, окно и необходимые библиотеки. После этого копию *pse::Game* можно получить функцией *pse::Game::i()*.

### Главный цикл

Логика игры и отрисовки описана в классах, наследованных от *pse::GameState*.

Для запуска главного цикла движа необходимо создать новый *pse::GameState* и передать его в *pse::Game*:

*pse::GameState\* gameState = new GameState();*

*pse::Game::i().push\_state(gameState);*

*pse::Game* содержит стек, состоящий из ссылок на различные *pse::GameState*. Для работы с ним используются функции *push\_state(pse::GameState\*), pop\_state(), set\_state(pse::GameState\*)*, соответствующие действиям над стеком.

Для запуска главного цикла используется функция *pse::Game::i().run()*. Главный цикл оперирует с верхним элементом стека.

### pse::GameState

*pse::GameState* содержит следующие чисто виртуальные методы:

* *void handleEvent(SDL\_Event\* event)*

Обработка событий клавиатуры/мыши

* *void update(float delta)*

Обновление состояния. *delta* – время, прошедшее с момента прошлого обновления

* *void draw()*

Обновление экрана

Для отрисовки используется структура *pse::RenderInfo*. Активный *pse::GameState* должен создать её и инициализировать поля framebufferDefault и framebufferUI ссылками на объекты типа *pse::Framebuffer*.

## Графика

### pse::RenderInfo

Структура, используемая для передачи данных при отрисовке графики.

Поля:

* *glm::mat4 projection*

Матрица проекции

* *glm::mat4 view*

Матрица вида

* *glm::vec2 position*

Позиция

* *glm::vec2 scale*

Размер

* *float rotation = 0*

Поворот

* *std::unique\_ptr<Framebuffer> framebufferDefault*

Framebuffer по умолчанию

* *std::unique\_ptr<Framebuffer> framebufferUI*

Framebuffer для элементов интерфейса

*framebufferUI* отрисовывается поверх *framebufferDefault*.

Объекты, принимающие *RenderInfo*, могут менять его для передачи данных в дочерние объекты.

**Примечание**: объекты, меняющие поле rotation, должны вернуть его в значение 0 после использования.

### pse::TextureManager

Класс *pse::TextureManager* служит для управления текстурами. Поддерживаются форматы bmp и png.

Методы:

* *pse::Texture::Ptr load(std::string filename)*

Загружает текстуру из файла *filename*. Если текстура с таким именем файла уже была загружена, будет возвращен указатель на уже существующую текстуру. При возникновении ошибки выбрасывает исключение.

* pse::Texture::Ptr create(unsigned int width, unsigned int height)

Создает пустую текстуру размера *width* x *height*. При возникновении ошибки выбрасывает исключение.

### pse::ShaderManager

Класс *pse::ShaderManager* служит для управления GLSL шейдерами.

Методы:

* *Shader::Ptr load(std::string vert, std::string frag)*

Создает шейдер из файлов *vert* и *frag*. При возникновении ошибки выбрасывает исключение.

### pse::ResourceManager

Класс *pse::ResourceManager* служит для управления менеджерами ресурсов. Синглтон.

Методы:

* *ShaderManager\* shader() const*

Возвращает *pse::ShaderManager*

* *TextureManager\* texture() const*

Возвращает *pse::TextureManager*

Также класс имеет макросы для вызова вышеописанных методов – *pse::ResourceTexture* и *pse::ResourceShader* соответственно.

### pse::Texture

Класс *pse::Texture* служит для управления конкретной текстурой. Поддерживаются форматы bmp и png.

Загрузка и создание текстуры:

*ResourceTexture->load(“path/to/texture.png”)*

Этот метод возвращает *pse::Texture::Ptr* (*std::shared\_ptr<pse::Texture>*) или выбрасывает исключение.

Методы *pse::Texture*:

* *void bind(GLuint index)*

Устанавливает текстуру как активную с соответствующим индексом

* *glm::vec2 scale() const*

Возвращает размер текстуры в пикселях

* *glm::vec2 scale() const*

Возвращает соотношение ширины текстуры к высоте текстуры

### pse::Shader

Класс *pse::Shader* служит для управления GLSL шейдерами.

Загрузка и создание GLSL шейдера:

*ResourceShader->load(“path/to/vertex\_shader.vert”, “path/to/fragment\_shader.frag”)*

Этот метод возвращает *pse::Shader::Ptr* (*std::shared\_ptr<pse::Shader>*) или выбрасывает исключение.

Методы *pse::Shader*:

* *void bind()*

Устанавливает данный шейдер как активный

* *GLint uniform(std::string uniform\_name)*

Возвращает идентификатор uniform-переменной в шейдере

* GLint uniform(std::string uniform\_name, ...)

Устанавливает значение uniform-переменной в шейдере (uniform перегружен для различных типов, поддерживаемых uniform-переменными - <https://www.opengl.org/sdk/docs/man/html/glUniform.xhtml>)

### pse::Vertex

Структура, используемая для описания вершин полигональной сетки.

Поля:

* *glm::vec3 position*

Координаты вершины

* *glm::vec2 uv*

Текстурные координаты вершины

### pse::Mesh

Класс, описывающий полигональную сетку.

*Mesh(std::vector<Vertex>& vertices, std::vector<GLuint>& indices)*

*vertices -* набор вершин. *indices* - индексы вершин.

Методы:

* *void drawElements() const*

Рисует полигональную сетку

* *GLuint getVAO() const*

Возвращает идентификатор Vertex Array Object текущего объекта

* *GLuint getVBO() const*

Возвращает идентификатор Vertex Buffer Object текущего объекта

* *GLuint getEBO() const*

Возвращает идентификатор Element Buffer Object текущего объекта

### pse::Framebuffer

Класс *pse::Framebuffer* служит для создания и управления фреймбуферами OpenGL.

* *pse::Framebuffer()*

Создание фреймбуфера с размером, равным размеру окна (также размер динамически изменяется при изменении размера окна)

* *Framebuffer(unsigned int width, unsigned int height, bool wh\_auto = false)*

Создание фреймбуфера с заданным размером, *wh\_auto* указывает, необходимо ли изменять размер фреймбуфера при изменении размера окна.

В случае, если невозможно создать новый фреймбуфер выкидывается исключение.

Методы *pse::Framebuffer*:

* *void reallocate(unsigned int width, unsigned int height)*

Изменяет размер фреймбуфера

* *void bind()*

Устанавливает фреймбуфер как активный

Также, *pse::Framebuffer* содержит структуру *Default*, представляющую стандартный фреймбуфер. *pse::Framebuffer::Default* реализует только метод *bind*.

### pse::Sprite

Класс *pse::Sprite* служит для управления спрайтами.

* *Sprite(const pse::Texture::Ptr& texture, int width, int height, int start, int end, glm::vec2 scale)*

Создание спрайта из текстуры, содержащей *width* спрайтов по ширине и *height* по высоте. Спрайт может менять кадры от кадра *start* до кадра *end*. Спрайт имеет размеры, указанные в *scale*.

* *Sprite(const pse::Texture::Ptr& texture, int width, int height, int start, int end, float scale)*

Создание спрайта из текстуры, содержащей *width* частей спрайта по ширине и *height* частей спрайта по высоте. Спрайт может менять кадры с кадра *start* до кадра *end*. Спрайт имеет отношение ширины к высоте, указанное в *scale*.

* *void tick()*

Обновление спрайта (смена кадров)

* *void draw(pse::RenderInfo\* renderInfo)*

Отрисовывает спрайт на текущем фреймбуфере с данными из *renderInfo*. Если установлен пользовательский шейдер *pse::Sprite:custom\_shader*, использует его, вместо стандартного шейдера спрайтов.

* *glm::vec2 scale() const*

Возвращает размер спрайта

* *void scale(glm::vec2)*

Устанавливает размер спрайта

* *void scale(float)*

Устанавливает соотношение сторон спрайта

### pse::RenderManager

Синглтон, управляющий процессом обновления экрана. Для получения копии этого класса используется функция *pse::Game::i().render().*

Методы:

* *void clear() const*

Очищает экран и фреймбуферы

* *void swap() const*

Закончить текущий кадр (поменять буферы местами)

* *void deferred(pse::RenderInfo\* renderInfo) const*

Объединить *renderInfo->framebufferDefault* и *renderInfo->framebufferUI* в *pse::Framebuffer::Default*

Таким образом, тело цикла обновления экрана выглядит следующим образом:

*pse::Game::i().render()->clear();*

*//Отрисовка объектов на экране*

*pse::Game::i().render()->deferred(&renderInfo);*

*pse::Game::i().render()->swap();*

### pse::WindowManager

Синглтон, управляющий окном, контекстом SDL2 и OpenGL. Для получения копии этого класса используется функция *pse::Game::i().window().*

Методы:

* *unsigned int width() const*

Возвращает ширину окна в пикселях

* *unsigned int height() const*

Возвращает высоту окна в пикселях

### pse::Font

Класс для создания строк для отрисовки на экран.

Методы:

* *pse::Font::String\* string(std::string, float size = 50)*

Создает новую строку для отрисовки с заданным размером шрифта. Возвращает указатель на класс *pse::Font::String*.

При инициализации игры инициализируется поле *pse::Font::Default*, содержащее стандартный шрифт (ссылку на *pse::Font*).

### pse::Font::String

Класс, описывающий строку для отрисовки на экран.

Методы:

* *void draw(pse::RenderInfo\* renderInfo)*

Отрисовывает строку на текущем фреймбуфере с данными из *renderInfo*.

Поля:

* *float size*

Задает размер шрифта.

### pse::Camera

Класс, служащий для управления ортогональной камерой.

*pse::Camera(float scale)*

*scale* - размер видимого пространства по высоте. (Ширина вычисляется по *ratio*, соотношению сторон)

Методы:

* *glm::mat4 getProjection(GLfloat ratio)*

Возвращает матрицу проекции для заданного ratio

* *glm::vec4 getViewport(GLfloat ratio)*

Возвращает матрицу обзора

Поля:

* *float scale*

Размер видимого пространства по высоте. (Ширина вычисляется по *ratio*, соотношению сторон)

* *glm::vec2 offset*

Смещение видимого пространства

## Таймеры

### pse::Timer

Класс, реализующий таймеры игрового цикла. Используется для отложенного или периодического выполнения неких функций.

*pse::Timer::Ptr pse::Timer::create(std::function<void()> callback, unsigned int duration, bool repeat, bool paused)*

Создает новый таймер и возвращает shared pointer на него.

*pse::Timer::Ptr pse::Timer::create(unsigned int duration, bool repeat, bool paused)*

Создает новый таймер без функции, вызываемой по завершении, и возвращает shared pointer на него.

*callback* – функция, вызываемая при завершении таймера

*duration* – длительность таймера

*repeat* – указывает, необходимо ли перезапускать таймер после завершения

*paused* – указывает, необходимо ли приостановить таймер сразу после создания

Методы:

* *void cancel()*

Отменить выполнение таймера

* *void pause()*

Приостановить выполнение таймера

* *void resume()*

Продолжить выполнение таймера

* *void toggle()*

Приостановить/продолжить выполнение таймера

* *void reset()*

Вернуть таймер в начальное состояние

* *unsigned int getDuration() const*

Возвращает длительность таймера

* *unsigned int getCurrent() const*

Возвращает текущий тик таймера

* *pse::Timer::timer\_state getState() const*

Возвращает состояние таймера (TIMER\_ACTIVE, TIMER\_PAUSE, TIMER\_FINISHED)

* *bool isFinished() const*

Проверяет, завершён ли таймер

* *bool isActive() const*

Проверяет, активен ли таймер

* *bool isPaused() const*

Проверяет, приостановлен ли таймер

## Интерполяция

### pse::Interpolator

Класс, используемый для интерполяции некоторых переменных по указанной функции на указанном интервале времени (в тиках игры).

*pse::Interpolator<T>::Interpolator(T\* target\_entity, float (\* interpolation)(float, float, float, float))*

Создает интерполятор для переменной типа *T* для её интерполяции по функции *interpolation*.

Возможные функции интерполяции:

* *pse::interp::Back*
* *pse::interp::Bounce*
* *pse::interp::Circ*
* *pse::interp::Linear*
* *pse::interp::Cubic*
* *pse::interp::Elastic*
* *pse::interp::Expo*
* *pse::interp::Quad*
* *pse::interp::Quart*
* *pse::interp::Quint*
* *pse::interp::Sine*

Методы:

* *void offset(T offset, unsigned ticks)*

Интерполяция переменной, для которой создан интерполятор, от текущего значения переменной до суммы текущего значения и *offset,* в течении *ticks* тиков игры.

* *void target(T target, unsigned ticks)*

Интерполяция переменной, для которой создан интерполятор, от текущего значения до значения *target,* в течении *ticks* тиков игры.

* *void cancel()*

Отменяет интерполяцию

* *pse::Interpolator::interp\_state state() const*

Возвращает состояние интерполяции (INTERP\_ACTIVE, INTERP\_FINISHED)

* *bool finished() const*

Проверяет, завершена ли интерполяция

### pse::InterpolatorQueue

Класс, используемый для создания очереди интерполяций переменной.

pse::*InterpolatorQueue<T>::InterpolatorQueue(T\* target\_entity, float (\* interpolation)(float, float, float, float))*

Создает очередь интерполяций для переменной типа *T* для её интерполяции по функции *interpolation*.

Методы:

* *void push\_offset(T offset, unsigned ticks)*

Добавляет в очередь интерполяцию переменной, для которой создан интерполятор, от текущего значения переменной до суммы текущего значения и *offset,* в течении *ticks* тиков игры.

* *void push\_target(T target, unsigned ticks)*

Добавляет в очередь интерполяцию переменной, для которой создан интерполятор, от текущего значения до значения *target,* в течении *ticks* тиков игры.

## Игра 0xff2

Непосредственно код игры состоит из классов, описывающих: сущности, игрока, врагов, пули, хитбоксы (коллизию игрока, врагов, пуль), текущее состояние игры, начальный экран, логику игры.

### Сущности, Entity

Класс, описывающий игровую сущность.

*Entity()*

Методы:

* *virtual void draw(RenderInfo\* renderInfo) const*

Отрисовка сущности

* *virtual void update(StateInfo\* stateInfo)*

Обновление состояния сущности

* *const glm::vec2& getPosition() const*

Возвращает позицию сущности в игровом пространстве

* *void setPosition(glm::vec2 \_pos)*

Устанавливает позицию сущности в игровом пространстве

* *void setRotation(float \_rotation)*

Устанавливает угол поворота сущности в игровом пространстве

* *Hitbox\* getHitbox() const*

Возвращает хитбокс (коллизию) сущности

### Игрок, Player

Класс, описывающий игрока. Наследует Entity и реализует его виртуальные методы.

Кроме наследованны, реализует следующие методы:

* *void handleEvent(SDL\_Event\* event)*

Служит для передачи игроку событий SDL2 (для обработки клавиатуры).

* *bool hit()*

Вызывается при столкновении игрока с врагом/вражеской пулей. Возвращает true, если возможно отнять жизнь у игрока

* *bool collision(Hitbox\* hitbox)*

Проверяет, произошло ли столкновение между хитбоксом игрока и переданным хитбоксом.

* *std::list<BulletInstance\*>\* getBullets()*

Возвращает список пуль игрока

* *unsigned short getLives() const*

Возвращает количество жизней игрока

* *Player::state\_enum getState() const*

Возвращает состояние игрока (PLAYER\_STATE\_ALIVE, PLAYER\_STATE\_HIT\_ANIMATION, PLAYER\_STATE\_DEATH\_ANIMATION, PLAYER\_STATE\_DEAD)

* *bool isAlive() const*

Возвращает true, если состояние игрока не равно PLAYER\_STATE\_DEAD

### Враг, Enemy

Класс, описывающий игрока. Наследует Entity и реализует его виртуальные методы.

Кроме наследованны, реализует следующие методы:

* *bool isBoss() const*

Возвращает true, если враг является боссом

* *Enemy::state\_enum state() const*

Возвращает состояние врага (ENEMY\_STATE\_ALIVE, ENEMY\_STATE\_DEATH\_ANIMATION, ENEMY\_STATE\_DEAD)

* *void kill()*

Убивает врага

### Пули, BulletType

Класс, описывающий тип пули. Есть возможность наследовать новый класс от данного, чтобы создать новый тип пули, имеющий особую отрисовку или особое поведение.

Методы:

* *void render(pse::RenderInfo\* renderInfo, BulletInstance\* bulletInstance)*

Метод, вызываемый для отрисовки конкретной пули, используя текущий тип

* *void update(StateInfo\* stateInfo, BulletInstance\* bulletInstance)*

Метод, используемый для обновления пули, используя текущий тип.

Поля:

* *std::unique\_ptr<Sprite> sprite*

Спрайт, используемый для отрисовки пуль данного типа

### Пули, BulletInstance

Класс, описывающий конкретную пулю.

Поля:

* *unsigned tick*

Тик пули. Увеличивается при каждом обновлении.

* *unsigned ttl*

Время жизни пули, в тиках

* *glm::vec2 pos*

Позиция пули в игровом пространстве

* *glm::vec2 vel*

Скорость пули

* *float angle*

Угол поворота пули

* *Hitbox\* hitbox*

Хитбокс пули

* *BulletType\* type*

Тип пули

### Пули, BulletHell

Класс, описывающий набор BulletInstance. Реализует “сборку мусора” среди пуль: удаляются пули, у которых *tick* превышает *ttl*, или которые вышли за пределы {-1000, -100} - {1000, 1000} в координатах игрового поля.

Методы:

* *void draw(pse::RenderInfo\* renderInfo) const*

Отрисовка всех пуль

* *void update(StateInfo\* stateInfo)*

Обновление всех пуль

* *void push(BulletInstance\* bulletInstance)*

Добавление новой пули в список

Поля:

* *std::list<BulletInstance\*> bullets*

Текущий список пуль

### Хитбокс, Hitbox

Класс, описывающий хитбокс в виде окружности с заданным радиусом.

*Hitbox(float radius)*

Методы:

* void draw(pse::RenderInfo\* renderInfo) const

Отрисовка хитбокса. **Примечание:** реализация отрисовки является крайне неэффективной, поэтому, для избежания падения производительности, отрисовку хитбоксов стоит использовать *только* для отладки.

* *bool collision(Hitbox& hitbox) const*

Проверяет коллизию между текущим хитбоксом и *hitbox*.

* *glm::vec2 getPosition() const*

Возвращает позицию хитбокса в игровом пространстве

* *void setPosition(glm::vec2 \_pos)*

Устанавливает позицию хитбокса в игровом пространтсве

* *float getRadius() const*

Возвращает радиус окружности хитбокса

* *void setRadius(float \_radius)*

Устанавливает радиус окружности хитбокса

### Состояние игры, StateInfo

Структура, используемая для хранения информации о состоянии игры и для передачи её в другие объекты.

Поля:

* *uint64\_t tick*

Тик игры

* *class Player\* player*

Ссылка на текущего игрока

### Начальный экран, GameStateStartScreen

Класс, наследующий pse::GameState и описывающий начальный экран. Содержит отрисовку фона, названия игры, надписи “Press (Enter)”, анимацию надписей, переход к *GameStateGame* по нажатию кнопки Enter, анимацию перехода к игре, выход из игры по нажатию кнопки Escape.

### Экран игры, GameStateGame

Класс, наследующий pse::GameState и описывающий экран игры. Содержит всю логику игры, включая коллизию между пулями и игроком или врагами, между игроком и врагами, подсчёт очков, управление врагами (используя отдельные классы *BulletPattern, InvadersFormation* и т.д.). Также реализует анимацию перехода к игре из начального экрана, перехода к начальному экрану из игры при нажатии кнопки Escape, перезапуск экрана игры при долгом нажатии кнопки R, анимацию перезапуска игры.

# Заключение

В данной работе была выполнена поставленная задача – реализация игры Space Invaders. Также, в процессе выполнения работы были в некоторой степени изучены OpenGL 3.1, GLSL (язык шейдеров OpenGL), SDL2 (библиотека, используемая для создания контекста OpenGL и обработки событий с клавиатуры), glm (для математических функций и матричных преобразований). Также был реализован некоторый “игровой движок”, позволяющий реализовывать игры, основанные на двумерной графике, таймерах и интерполяции, в том числе и игру Space Invaders.

# Приложение

## Скриншоты







# Рецензия

# Отзыв руководителя