

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ «МИСИС»

Институт информационных технологий и компьютерных наук

Кафедра инженерной кибернетики

Курсовая работа

по дисциплине «Технологии программирования»

на тему

«Создание движка для разработки 2D игр»

Выполнил:

студент 2-го курса,

гр. БПМ-21-3 Привалов М.А.

Проверил:

доцент, к.т.н. Полевой Д.В.

Москва, 2022

Содержание

Описание задачи.....	3
Техническое задание	4
Проектирование.....	5
1. Графическая компонента:.....	5
2. Компонента для работы с ресурсами	27
3. Физическая компонента	34
4. Компонента программного интерфейса	37
5. Игровой цикл	45
Инструкция по установке.....	46
Инструкция по использованию движка.....	48
1) Подготовка ресурсов	48
2) Создание уровня	52
3) Создание танка и реализация его стрельбы.....	63
4) Класс игры	69

Описание задачи

Спроектировать и создать как можно более гибкий игровой движок для разработки 2D игр, используя язык программирования с++ и графической библиотекой OpenGL.

Данный продукт экономит время разработчикам игр, так как им не нужно будет реализовывать базовые функции игры такие как отрисовка на экране, считывание спрайтов с файла, присваивание объектам физических свойств (твёрдость, скорость и т.д) и работа со временными событиями/аномалиями/эффектами.



Техническое задание

В движке должны быть реализованы:

1. **Графическая компонента (RenderEngine).** Должна реализовывать функции, позволяющие покадрово отрисовывать игровой процесс, объединять шейдеры и текстуры, создавая спрайты для игровых объектов; давать возможность создавать покадровую анимацию.
2. **Компонента для работы с ресурсами (ResourceManager).** Должна обрабатывать ресурсы, считывать текстуры с png/jpeg – файлов, считывать разметку для ресурсов с json файлов и объединять эти данные с графическими данными; создание шейдеров, текстур.
3. **Физическая компонента (PhysicsEngine).** Должна реализовывать набор функций, придающих игровым объектам физические свойства: скорость, твёрдость; обрабатывать взаимодействия объектов друг с другом.
4. **Компонента программного интерфейса (GameObjectInterface, LevelInterface).** Должна обеспечивать пользователя базовыми методами, реализующими функционал движка, которые пользователь при желании смог бы переопределить.
5. **Игровой цикл (main.cpp).** Файл, соединяющий воедино написанный пользователем код и внутренние функции движка и заставляющий всё это работать каждый кадр, пока игру не выключат.

Проектирование

1. Графическая компонента:

Шейдер (англ. Shader) — это программа для одной из ступеней графического конвейера, используемая в трёхмерной графике для определения окончательных параметров объекта или изображения.

Шейдеры необходимы для разметки в пространстве точек, которым нужно придать определённый цвет или наложить текстуру, то есть, создать картинку.

В OpenGL шейдеры программируются на C подобном языке GLSL. Он адаптирован для использования в графике и предоставляет функции работы с векторами и матрицами.

Существуют вершинные шейдеры, отвечающие за точки-вершины объекта, который нужно отрисовать и фрагментные шейдеры, отвечающие за цвет, которым нужно заполнить пиксели, ограниченные вершинами.

Для каждого игрового объекта будет создаваться свой шейдер по определённому шаблону, который нужно написать и для вершинного и для фрагментного шейдера.

```
#version 460 //Версия OpenGL
layout(location = 0) in vec2 vertex_position; // расположение (location) для
атрибута вершины

layout(location = 1) in vec2 texture_position; //расположение (location) для
атрибута текстур

out vec2 texturePoints; //Выходные данные для фрагментного шейдера – координаты
для наложения текстуры
// in - входные данные, out - выходные
uniform mat4 modelMat; //матрица модели
uniform mat4 projectionMat; //матрица проекция
//Эти матрицы нужны для работы с системами координат в OpenGL
uniform float layer; //Слой текстуры

void main()
{
    texturePoints = texture_position;
    gl_Position = projectionMat*modelMat*vec4(vertex_position, layer, 1.0);
    //Вычисление положения объекта в системе координат opengl
};
```

Шаблон вершинного шейдера

```
#version 460
in vec2 texturePoints; //Входные данные для фрагментного шейдера – координаты
для наложения текстуры
out vec4 frag_color; //Выходные данные о цвете пикселей

uniform sampler2D textures; //Текстура для фрагментного шейдера

void main() {
```

```

    frag_color = texture(textures, texturePoints); //сэмплирование цвета
текстуры
    if(frag_color.rgb == vec3(0.)) //Игнорирование черного цвета на текстуре для
создания прозрачности
    {
        discard;
    } };

```

Шаблон фрагментарного шейдера

Данные программы шейдеров стоит вынести в отдельные файлы и считывать их уже в коде на с++. (Файлы \res\shaders\fragmentSprite.txt и \res\shaders\vertexSprite.txt)

Для работы с этими шейдерами понадобится программа, позволяющая для каждого игрового объекта создавать эти шейдеры, удалять их и т.д.

Класс `RenderEngine::ShaderManager`

Класс, обеспечивающий создание шейдеров, их использование и уничтожение. Входит в пространство имён графической компоненты движка "RenderEngine". glad обеспечивает создание переменных таких типов, которые являются "понятными" для OpenGL. `mat4x4` позволяет работать с трансформационными матрицами для передвижения отрисованных объектов

```
#include <ShaderManager.h>
```

Открытые члены

- **ShaderManager** (const std::string &vertexShader, const std::string &fragmentShader)
- **~ShaderManager** ()
- void **UseShader** () const
- void **SetInt** (const std::string &name, const GLint value)
- void **SetFloat** (const std::string &name, const GLfloat value)
- void **SetMatrix4x4** (const std::string &name, const glm::mat4 &matrix)
- bool **IsCompiled** () const
- **ShaderManager** ()=delete
- **ShaderManager** (const **ShaderManager** &)=delete
- **ShaderManager** & **operator=** (const **ShaderManager** &)=delete
- **ShaderManager** & **operator=** (**ShaderManager** &&shaderManager) noexcept
- **ShaderManager** (**ShaderManager** &&shaderManager) noexcept

Подробное описание

Класс, обеспечивающий создание шейдеров, их использование и уничтожение. Входит в пространство имён графической компоненты движка "RenderEngine". glad обеспечивает создание переменных таких типов, которые являются "понятными" для OpenGL. `mat4x4` позволяет работать с трансформационными матрицами для передвижения отрисованных объектов

Конструктор(ы)

RenderEngine::ShaderManager::ShaderManager (const std::string & vertexShader, const std::string & fragmentShader)

Конструктор класса. Создаёт вершинные и фрагментные шейдеры и связывает их в одну шейдерную программу, готовую для использования. После чего удаляет ненужные больше вершинные и фрагментные шейдеры.

Аргументы

<i>vertexShader, fragmentShader</i>	- ссылки на адреса файлов, содержащих интерфейсы шейдеров, написанные на GLSL Реализация метода:
-------------------------------------	--

```
GLuint vertexShaderId; // Создаём id вершинного шейдера типа инт, понятного glsl
- шейдерному языку, похожему на C
if (!CreateShader(vertexShader, GL_VERTEX_SHADER, vertexShaderId)) // Если метод
создания вершинного шейдера выдал ошибку, т.е вернул фолс, сообщаем пользователю,
что его вершинный шейдер не создался. Как правило, такая ошибка может возникнуть
из-за не поладок в шейдерной программе
{
    std::cerr << "VERTEX SHADER compile-time error" << std::endl;
    return;
}

GLuint fragmentShaderId; // Аналогично, как и с вершинным шейдером проделываем ту
же операцию создания идентификатора для фрагментного шейдера
if (!CreateShader(fragmentShader, GL_FRAGMENT_SHADER, fragmentShaderId)) //
Аналогично, как и с вершинным шейдером проделываем ту же операцию проверки
успешности создания фрагментного шейдера
{
    std::cerr << "FRAGMENT SHADER compile-time error" << std::endl;
    glDeleteShader(vertexShaderId);
    return;
}

id = glCreateProgram(); // Создаем шейдерную программу и получаем её
идентификатор.
glAttachShader(id, vertexShaderId); // Присоединяем вертексный шейдер к шейдерной
программе с идентификатором id
glAttachShader(id, fragmentShaderId); // Присоединяем фрагментный шейдер к
шейдерной программе с идентификатором id
glLinkProgram(id);

GLint success; // По хорошему надо проверить, всё ли нормально сработало, этим и
займёмся. Создаём переменную, по которой мы потом поймём, сработало ли всё как
надо или нет
glGetProgramiv(id, GL_LINK_STATUS, &success); // Узнаём, слинковалась ли шейдерная
программа с идентификатором id, результат передаем в уже созданную переменную
"успех"
if (!success) // Если удача сегодня не на нашей стороне, получаем сообщение с
ошибкой линковки
{
    GLchar infoLog[1024];
    glGetShaderInfoLog(id, 1024, nullptr, infoLog);
    std::cerr << "Link error: " << infoLog << std::endl;
}
else // Если же всё ок, то меняем isCompiled с false на true, знание о созданном
шейдере нам ещё ой как понадобится. Но это будет не скоро. Или скоро. Смотря,
какой задокументированный класс ты откроешь.
{
    isCompiled = true;
}

glDeleteShader(vertexShaderId); // Кстати, наши вершинные шейдеры уже выполнили
свою миссию - слинковались с шейдерной программой, чтобы породить на свет ещё один
шейдер. Да, он молодец, но теперь он бесполезен и только занимает память. Удаляем
его.
glDeleteShader(fragmentShaderId); // Фрагментного шейдера это тоже касается.
Удаляем.
```

RenderEngine::ShaderManager::~ShaderManager ()

Деструктор класса - удаляет шейдерную программу

RenderEngine::ShaderManager::ShaderManager ()[delete]

Запрещаем конструктор без параметров, так как это не имеет смысла.

RenderEngine::ShaderManager::ShaderManager (const ShaderManager &)[delete]

Запрещаем копирование менеджера шейдера в другой менеджер, так как у нас получатся два объекта с одним и тем же id и когда программа будет удаляться, выйдет неожиданный результат.

RenderEngine::ShaderManager::ShaderManager (ShaderManager && shaderManager) [noexcept]

Конструктор копирования объекта, который скоро удалится тоже разрешён, так как казусов с id шейдера не возникнет

Аргументы

<i>shaderManager</i>	- объект, передающийся по move ссылке
----------------------	---------------------------------------

```
id = shaderManager.id; //Делаем новый id шейдерной программы, взятый у копируемого менеджера шейдеров
isCompiled = shaderManager.isCompiled; //Передаём текущему isCompiled значение копируемого менеджера шейдеров

shaderManager.id = 0; //Отключаем копируемый шейдер менеджер
shaderManager.isCompiled = false;
```

Методы

bool RenderEngine::ShaderManager::IsCompiled () const [inline]

Метод для проверки успешности работы с шейдерами

Возвращает

isCompiled - приватное поле класса

ShaderManager & RenderEngine::ShaderManager::operator= (const ShaderManager &)[delete]

Запрещаем присваивание менеджера шейдера в другой менеджер, так как у нас получатся два объекта с одним и тем же id и когда программа будет удаляться, возникнет казус невероятный.

ShaderManager & RenderEngine::ShaderManager::operator= (ShaderManager && shaderManager) [noexcept]

Присваивать одному экземпляру менеджера шейдера другого, который скоро удалится, можно, так как не возникнет проблем с id, он будет уникальным.

Аргументы

<i>shaderManager</i>	- объект, передающийся по move ссылке
----------------------	---------------------------------------

```
glDeleteProgram(id); //Удаляем прошлую шейдерную программу
id = shaderManager.id; //Делаем новый id шейдерной программы, взятый у присваиваемого менеджера шейдеров
isCompiled = shaderManager.isCompiled; //Передаём текущему isCompiled значение присваиваемого менеджера шейдеров

shaderManager.id = 0; //Отключаем присваиваемый шейдер менеджер
shaderManager.isCompiled = false;
return *this; //Возвращаем новоиспеченный объект
```




void RenderEngine::ShaderManager::SetFloat (const std::string & *name*, const GLfloat *value*)

Используется при работе с текстурами для того, чтобы установить позицию текстурного блока в uniform sampler (текстурный объект в файле). Устанавливая их через glUniform1f мы будем уверены, что uniform sampler соотносится с правильным текстурным блоком.

Аргументы

<i>name</i>	- ссылка на имя шейдерной программы
<i>value</i>	- номер текстурного блока

void RenderEngine::ShaderManager::SetInt (const std::string & *name*, const GLint *value*)

Используется при работе с текстурами для того, чтобы установить позицию текстурного блока в uniform sampler (текстурный объект в файле). Устанавливая их через glUniform1i мы будем уверены, что uniform sampler соотносится с правильным текстурным блоком.

Аргументы

<i>name</i>	- ссылка на имя шейдерной программы
<i>value</i>	- номер текстурного блока

void RenderEngine::ShaderManager::SetMatrix4x4 (const std::string & *name*, const glm::mat4 & *matrix*)

Метод передачи трансформационной матрицы в шейдер. Нужен для того, чтобы объект двигался

Аргументы

<i>name</i>	- имя матрица
<i>matrix</i>	- трансформационная матрица

void RenderEngine::ShaderManager::UseShader () const

Используем шейдерную программу для дальнейшей отрисовки объектов

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Renderer/ShaderManager.h
- GameEngineAndGame/src/Renderer/ShaderManager.cpp

Теперь эти шейдеры нужно применить, то есть создать классы, позволяющие накладывать текстуры с помощью шейдеров, создавая спрайты. Но для начала текстуру тоже нужно создать, используя методы OpenGL. Для этого есть класс TextureManager

Методы для работы с OpenGL предоставляют библиотеки **glad** – функции OpenGL, **glfw** – для отрисовки окна, **glm** – математическая компонента OpenGL

Класс RenderEngine::TextureManager

Класс для работы с текстурами

```
#include <TextureManager.h>
```

Классы

struct **Tile** Структура, хранящая координаты текстуры (верхняя правая точка, нижняя левая точка)

Открытые члены

- void **AddTile** (std::string name, const glm::vec2 &leftBottom, const glm::vec2 &rightTop)
- const **Tile** & **GetTile** (const std::string &name) const
- unsigned int **GetWidth** () const
- unsigned int **GetHeight** () const
- void **Bind** () const
- **TextureManager** (const GLuint width, GLuint height, const unsigned char *textFromFile, const unsigned int channels=4, const GLenum filter=GL_LINEAR, const GLenum wrapMode=GL_CLAMP_TO_EDGE)
- **TextureManager** (const **TextureManager** &)=delete
- **TextureManager** & **operator=** (const **TextureManager** &)=delete
- **TextureManager** & **operator=** (**TextureManager** &&texture)
- **TextureManager** (**TextureManager** &&texture)

Подробное описание

Класс для работы с текстурами

Конструктор(ы)

RenderEngine::TextureManager::TextureManager (const GLuint *width*, GLuint *height*, const unsigned char * *textFromFile*, const unsigned int *channels* = 4, const GLenum *filter* = GL_LINEAR, const GLenum *wrapMode* = GL_CLAMP_TO_EDGE)

Конструктор класса, генерирующий, связывающий и накладывающий текстуру

Аргументы

<i>width,height</i>	- ширина и высота текстуры
<i>textFromFile</i>	- данный о текстуре, считанные из файла
<i>channels</i>	- цветовые каналы

<i>filter</i>	- фильтр отрисовки
<i>wrapMode</i>	- режим свёртки Реализация:

```

switch (channels)
{
case 4:
    mode = GL_RGBA;
    break;
case 3:
    mode = GL_RGB;
    break;
default:
    mode = GL_RGBA;
    break;
}

glGenTextures(1, &id);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, id);
glTexImage2D(GL_TEXTURE_2D, 0, mode, this->width, this->height, 0, mode,
GL_UNSIGNED_BYTE, data);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrapMode);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrapMode);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, filter);
glGenerateMipmap(GL_TEXTURE_2D);

glBindTexture(GL_TEXTURE_2D, 0);

```

RenderEngine::TextureManager::TextureManager (TextureManager && texture)

Конструктор move-копирования Реализация: id = texture.id; texture.id = 0; mode = texture.mode; width = texture.width; height = texture.height;

Методы

void RenderEngine::TextureManager::AddTile (std::string name, const glm::vec2 & leftBottom, const glm::vec2 & rightTop)

Добавляет текстуру с указанными параметрами и именем

Аргументы

<i>name</i>	- имя текстуры
<i>leftBottom, rightTop</i>	- координаты ключевых точек текстуры Реализация:

```

tiles.emplace(std::move(name), Tile(leftBottom, rightTop));

```

void RenderEngine::TextureManager::Bind () const

Метод связывающий текстуру с текущим id к активному текстурному блоку

const TextureManager::Tile & RenderEngine::TextureManager::GetTile (const std::string & name) const

Получаем данные текстуры с полученным именем. Если такой нет, возвращает дефолтную текстуру с нулевыми параметрами

Аргументы

<i>name</i>	- имя текстур Реализация:
<pre>auto it = tiles.find(name); if (it != tiles.end()) { return it->second; } const static Tile defaultTile; return defaultTile;</pre>	

TextureManager & RenderEngine::TextureManager::operator= (TextureManager && texture)

Переопределение move-оператора присваивания Реализация: glDeleteTextures(1, &id); id = texture.id; texture.id = 0; mode = texture.mode; width = texture.width; height = texture.height; return *this;

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Renderer/TextureManager.h
- GameEngineAndGame/src/Renderer/TextureManager.cpp

Структура **RenderEngine::TextureManager::Tile**

Структура, хранящая координаты текстуры (верхняя правая точка, нижняя левая точка)

```
#include <TextureManager.h>
```

Открытые члены

- **Tile** (const glm::vec2 &leftBottom, const glm::vec2 &rightTop)

Открытые атрибуты

- glm::vec2 **leftBottom**
 - glm::vec2 **rightTop**
-

Подробное описание

Структура, хранящая координаты текстуры (верхняя правая точка, нижняя левая точка)

Объявления и описания членов структуры находятся в файле:

- GameEngineAndGame/src/Renderer/TextureManager.h

Текстур и шейдеров всё ещё не достаточно, чтобы создать почти что полноценный игровой объект – спрайт. Нужны ещё такие вместилища информации, чтобы один раз записать туда данные об объекте (шейдерах и текстурах) и больше не тратить мощности на их высчитывание. Эти вместилища в OpenGL – **буферы**.

Объекты буфера - это объекты OpenGL, которые хранят массив неформатированной памяти, выделенной контекстом OpenGL (он же графический процессор). Они могут использоваться для хранения данных вершин, данных пикселей, полученных из изображений или фреймбуфера, и множества других вещей.

В движке реализованы сразу несколько классов буферов, каждый отвечающий за свою задачу.

Класс `RenderEngine::IndexBuffer`

Зачастую в играх разработчики хотят придавать объектам разные формы, а не только треугольные. Однако OpenGL умеет отрисовывать только треугольники, линии и точки. Поэтому, нужно отрисовать так много треугольников, чтобы из них получилось что-то не треугольное. Например, чтобы вывести на экран квадрат достаточно нарисовать два треугольника. Но чем больше треугольников мы рисуем, тем больше вершин нам придётся передавать в буферный объект, а следовательно, в память. И самое страшное - эти вершины будут часто повторяться. К счастью, разработчики OpenGL создали для таких случаев `Index Buffer Object` - это буфер, вроде VBO, но он хранит индексы, которые OpenGL использует, чтобы решить какую вершину отрисовать. Это называется отрисовка по индексам (`indexed drawing`) и является решением вышеуказанной проблемы.

```
#include <IndexBuffer.h>
```

Открытые члены

- `IndexBuffer ()`
- `~IndexBuffer ()`
- `IndexBuffer (const IndexBuffer &)=delete`
- `IndexBuffer & operator= (const IndexBuffer &)=delete`
- `IndexBuffer & operator= (IndexBuffer &&indexBuffer) noexcept`
- `IndexBuffer (IndexBuffer &&indexBuffer) noexcept`
- `void InitializeBuffer (const void *data, const unsigned int elementsCount)`
- `void BindBuffer () const`
- `void UnBindBuffer () const`
- `unsigned int GetElementsCount () const`

Подробное описание

Зачастую в играх разработчики хотят придавать объектам разные формы, а не только треугольные. Однако OpenGL умеет отрисовывать только треугольники, линии и точки. Божечки, дак что же делать? Правильно, отрисовать так много треугольников, чтобы из них получилось что-то не треугольное. Например, чтобы вывести на экран квадрат

достаточно нарисовать два треугольника. Но чем больше треугольников мы рисуем, тем больше вершин нам придётся передавать в буферный объект (напомню, узнать что это можно в интернете, я итак много написал), а следовательно, в память. И самое страшное - эти вершины будут часто повторяться. К счастью, разработчики OpenGL создали для таких случаев Index Buffer Object - это буфер, вроде VBO, но он хранит индексы, которые OpenGL использует, чтобы решить какую вершину отрисовать. Это называется отрисовка по индексам (indexed drawing) и является решением вышеуказанной проблемы.

Так в примере с квадратом нам потребуется хранить лишь 4 вершины, а не 6

Так вот, этот класс как раз - таки и предназначен для создания этого буфера и работы с ним. Он будет необходим при рендере объектов.

Конструктор(ы)

RenderEngine::IndexBuffer::IndexBuffer ()

Конструктор класса. В нём мы разве что зададим дефолтные значение полей id и elementsCount. Всё приравняем к нулю

RenderEngine::IndexBuffer::~~IndexBuffer ()

Деструктор класса. Здесь мы удаляем буфер.

RenderEngine::IndexBuffer::IndexBuffer (const IndexBuffer &) [delete]

Запрещаем копирование класса в другой класс, так как у нас получатся два объекта с одним и тем же id и когда программа будет удаляться, выйдет казус неимоверный.

RenderEngine::IndexBuffer::IndexBuffer (IndexBuffer && indexBuffer) [noexcept]

Конструктор копирования объекта, который скоро удалится тоже разрешён, так как казусов с id буфера не возникнет Присваивает новому классу данные старого, а стрый класс очищает.

Методы

void RenderEngine::IndexBuffer::BindBuffer () const

Метод, связывающий данный индексный буфер с объектом буфера (IBO с VBO)

unsigned int RenderEngine::IndexBuffer::GetElementsCount () const [inline]

Метод, возвращающий количество вершин-индексов.

Возвращает

elementsCount - количество вершин-индексов

void RenderEngine::IndexBuffer::InitializeBuffer (const void * data, const unsigned int elementsCount)

Создаём индексный буфер, привязываем его к объекту буфера и копируем индексы в буфер

Аргументы

<i>data</i>	- индексы, которые мы хотим передать
<i>elementsCount</i>	- количество индексов

```
this->elementsCount = elementsCount; //присваиваем нужное количество элементов в поле класса
```

```
glGenBuffers(1, &id); //Создаём индексный буффер
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, id); //Связываем его с объектом буффера (IBO
с VBO)
glBufferData(GL_ELEMENT_ARRAY_BUFFER, elementsCount * sizeof(GLuint), data,
GL_STATIC_DRAW); //Копируем данные в объект буффера
```

IndexBuffer & RenderEngine::IndexBuffer::operator= (const IndexBuffer &) [delete]

Запрещаем присваивание класса в другой класс, так как у нас получатся два объекта с одним и тем же id и когда программа будет удаляться, возникнет непредвиденный момент.

IndexBuffer & RenderEngine::IndexBuffer::operator= (IndexBuffer && indexBuffer) [noexcept]

Присваивать одному экземпляру класса другого, который скоро удалится, можно, так как не возникнет проблем с id, он будет уникальным. Присваивает новому классу данные старого, а старый класс очищает.

void RenderEngine::IndexBuffer::UnBindBuffer () const

Метод, отвязывающий индексный буффер путём передачи ему нулевого индекса

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Renderer/IndexBuffer.h
- GameEngineAndGame/src/Renderer/IndexBuffer.c

Класс `RenderEngine::VertexArray`

Класс, описывающий массив вершин. Похож на **`VertexBuffer`** и **`IndexBuffer`**. Вообще VAO, т.е **`VertexArray`**, нужен для того, чтобы настроить атрибуты лишь единожды и передать их в VAO.

```
#include <VertexArray.h>
```

Открытые члены

- **`VertexArray`** (const **`VertexArray`** &)=delete
- **`VertexArray & operator=`** (const **`VertexArray`** &)=delete
- **`VertexArray & operator=`** (**`VertexArray`** &&vertexArray) noexcept
- **`VertexArray`** (**`VertexArray`** &&vertexArray) noexcept
- void **`BindBuffer`** () const
- void **`UnBindBuffer`** () const
- void **`AddVertexBuffer`** (const **`VertexBuffer`** &vertexBuffer, const **`VertexBufferLayout`** &layout)

Подробное описание

Класс, описывающий массив вершин. Похож на **`VertexBuffer`** и **`IndexBuffer`**. Вообще VAO, т.е **`VertexArray`**, нужен для того, чтобы настроить атрибуты лишь единожды и передать их в VAO.

Объявления и описания членов классов находятся в файлах:

- `GameEngineAndGame/src/Renderer/VertexArray.h`
- `GameEngineAndGame/src/Renderer/VertexArray.cpp`

Класс `RenderEngine::VertexBuffer`

Данный класс нужен для работы с объектами вершинного буфера, которые могут хранить большое количество вершин в памяти GPU. Преимущество использования таких объектов буфера, что мы можем посылать в видеокарту большое количество наборов данных за один раз, без необходимости отправлять по одной вершине за раз. Функционал данного класса полностью совпадает с функционалом **IndexBuffer**, только некоторые команды и параметры OpenGL отличаются.

```
#include <VertexBuffer.h>
```

Открытые члены

- **VertexBuffer** (const **VertexBuffer** &)=delete
- **VertexBuffer** & operator= (const **VertexBuffer** &)=delete
- **VertexBuffer** & operator= (**VertexBuffer** &&vertexBuffer) noexcept
- **VertexBuffer** (**VertexBuffer** &&vertexBuffer) noexcept
- void **InitializeBuffer** (const void *data, const unsigned int size)
- void **UpdateBuffer** (const void *data, const unsigned int size) const
- void **BindBuffer** () const
- void **UnBindBuffer** () const

Подробное описание

Данный класс нужен для работы с объектами вершинного буфера, которые могут хранить большое количество вершин в памяти GPU. Преимущество использования таких объектов буфера, что мы можем посылать в видеокарту большое количество наборов данных за один раз, без необходимости отправлять по одной вершине за раз. Функционал данного класса полностью совпадает с функционалом **IndexBuffer**, только некоторые команды и параметры OpenGL отличаются.

Объявления и описания членов классов находятся в файлах:

- `GameEngineAndGame/src/Renderer/VertexBuffer.h`
- `GameEngineAndGame/src/Renderer/VertexBuffer.cpp`

Класс `RenderEngine::VertexBufferLayout`

Данный класс позволяет более компактно упаковать в памяти данные об объекте, который нужно нарисовать, путём создания одного буферного объекта из веринного буфера и буфера с цветами. Таким образом в одном буфере по порядку будут размещены данные позиция,цвет,позиция,цвет и т.д

```
#include <VertexBufferLayout.h>
```

Открытые члены

- `VertexBufferLayout ()`
- `void ReserveElements (const size_t count)`
- `void AddElementLayoutFloat (const unsigned int count, const bool isDoNormalize)`
- `unsigned int GetStride () const`
- `const std::vector< VertexBufferLayoutElement > & GetLayoutElements () const`

Подробное описание

Данный класс позволяет более компактно упаковать в памяти данные об объекте, который нужно нарисовать, путём создания одного буферного объекта из веринного буфера и буфера с цветами. Таким образом в одном буфере по порядку будут размещены данные позиция,цвет,позиция,цвет и т.д

Конструктор(ы)

`RenderEngine::VertexBufferLayout::VertexBufferLayout ()`

Конструктор, в котором будет проинициализирован шаг между элементами, равный размеру типа элемента в байтах.

Методы

`void RenderEngine::VertexBufferLayout::AddElementLayoutFloat (const unsigned int count, const bool isDoNormalize)`

Добавляет элемент типа float в layoutElements

Аргументы

<i>count</i>	- параметр, необходимый для вычисления размера элемента в байтах
<i>isDoNormalize</i>	- параметр, нужный для <code>glVertexAttribPointer</code> , но не нам. В двмжве он всегда false Реализация:

```
unsigned int elementSize = count * static_cast<unsigned int>(sizeof(GLfloat));  
//Вычисляем размер элемента  
layoutElements.push back({ static_cast<GLint>(count), GL_FLOAT, isDoNormalize,  
elementSize}); //Помушаем элемент в вектор  
elementStride += layoutElements.back().size; //Увеличиваем шаг элемента на  
количество вершин в элементе, чтобы обшагивать их.
```

**const std::vector< VertexBufferLayoutElement > &
RenderEngine::VertexBufferLayout::GetLayoutElements () const [inline]**

Функция, возвращающая вектор элементов буфера

unsigned int RenderEngine::VertexBufferLayout::GetStride () const [inline]

Функция, возвращающая шаг между элементами

void RenderEngine::VertexBufferLayout::ReserveElements (const size_t count)

Функция, резервирующая в векторе элементов буфера нужное количество мест

Аргументы

<i>count</i>	- нужное количество мест
--------------	--------------------------

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Renderer/VertexBufferLayout.h
- GameEngineAndGame/src/Renderer/VertexBufferLayout.cpp

Структура `RenderEngine::VertexBufferLayoutElement`

эта структура хранит все необходимые элементы создаваемого нами буфера, которые будут передаваться в `glVertexAttribPointer` - функцию-интерпритатор вершинных данных

```
#include <VertexBufferLayout.h>
```

Открытые атрибуты

- `GLint count`
Количество элементов
- `GLenum type`
Их тип
- `GLboolean isDoNormalize`
параметр, нужный для `glVertexAttribPointer`, но не нам. В двмжве он всегда false
- `unsigned int size`
Размер элемента в байтах

Подробное описание

эта структура хранит все необходимые элементы создаваемого нами буфера, которые будут передаваться в `glVertexAttribPointer` - функцию-интерпритатор вершинных данных

Объявления и описания членов структуры находятся в файле:

- `GameEngineAndGame/src/Renderer/VertexBufferLayout.h`

Данного функционала достаточно, чтобы создавать спрайты и анимацию из них.

Класс `RenderEngine::Sprite`

Класс, отвечающий за наложение текстуры в нужном месте, нужного размера, в нужное время. По сути, это графическая компонента игрового объекта

```
#include <Sprite.h>
```

Классы

`struct FrameParams` Открытые члены

- `Sprite` (`std::shared_ptr< TextureManager > pTexture`, `std::string initialTile`, `std::shared_ptr< ShaderManager > pShaderManager`)
- `Sprite` (`const Sprite &`)=delete
- `Sprite & operator=` (`const Sprite &`)=delete
- `void Render` (`const glm::vec2 &position`, `const glm::vec2 &size`, `const float rotation`, `const float layer=0`, `const size_t frameId=0`) const
- `void InsertFrames` (`std::vector< FrameParams > framesParams`)
- `double GetFrameDuration` (`const size_t frameId`) const
- `size_t GetFramesCount` () const

Защищенные данные

- `std::shared_ptr< TextureManager > pTexture`
- `std::shared_ptr< ShaderManager > pShaderManager`
- `VertexArray vertexArray`
- `VertexBuffer vertexCoordsBuffer`
- `VertexBuffer textureCoordsBuffer`
- `IndexBuffer indexBuffer`
- `std::vector< FrameParams > framesParams`
- `size_t lastFrameId`

Подробное описание

Класс, отвечающий за наложение текстуры в нужном месте, нужного размера, в нужное время. По сути, это графическая компонента игрового объекта

Конструктор(ы)

`RenderEngine::Sprite::Sprite` (`std::shared_ptr< TextureManager > pTexture`, `std::string initialTile`, `std::shared_ptr< ShaderManager > pShaderManager`)

Конструктор спрайта

Аргументы

<code>pTexture</code>	- умный указатель на экземпляр класса менеджера текстур
<code>initialTile</code>	- имя текстуры, с которой инициализируется игровой объект
<code>pShaderManager</code>	- указатель на экземпляр класса менеджера шейдеров
<code>r</code>	Реализация:

```
const GLfloat vertexCoords[] = { //Здесь храним координаты вершин для вершинного шейдера. Это координаты относительно объекта
```

```

    0.f, 0.f,
    0.f, 1.f,
    1.f, 1.f,
    1.f, 0.f
};

auto tile = this->pTexture->GetTile(std::move(initialTile)); // получаем

const GLfloat textureCoords[] = { //Здесь храним координаты вершин для текстуры
    tile.leftBottom.x, tile.leftBottom.y,
    tile.leftBottom.x, tile.rightTop.y,
    tile.rightTop.x,   tile.rightTop.y,
    tile.rightTop.x,   tile.leftBottom.y,
};

const GLuint indices[] = { //Здесь храним индексы для индексного буфера
    0, 1, 2,
    2, 3, 0
};

vertexCoordsBuffer.InitializeBuffer(vertexCoords, 2 * 4 * sizeof(GLfloat));
//Инициализируем объект вертексного буфера
VertexBufferLayout vertexCoordsLayout; //Создаём экземпляр объекта
VertexBufferLayout для более компактного хранения данных
vertexCoordsLayout.AddElementLayoutFloat(2, false); //Добавляем в
vertexCoordsLayout элементы вершин
vertexArray.AddVertexBuffer(vertexCoordsBuffer, vertexCoordsLayout); //Добавляем в
массив вершин объект вертексного буфера

//то же самое делаем с координатами текстур
textureCoordsBuffer.InitializeBuffer(textureCoords, 2 * 4 * sizeof(GLfloat));
VertexBufferLayout textureCoordsLayout;
textureCoordsLayout.AddElementLayoutFloat(2, false);
vertexArray.AddVertexBuffer(textureCoordsBuffer, textureCoordsLayout);

//Инициализируем индексный буфер для индексов
indexBuffer.InitializeBuffer(indices, 6);

//Отвязываем ставшие ненужными vertexArray и indexBuffer
vertexArray.UnBindBuffer();
indexBuffer.UnBindBuffer();

```

Методы

double RenderEngine::Sprite::GetFrameDuration (const size_t *frameId*) const

Получаем данные о направлении объекта

size_t RenderEngine::Sprite::GetFramesCount () const

Получаем данные о количестве кадров

void RenderEngine::Sprite::InsertFrames (std::vector< FrameParams > *framesParams*)

Переносим данные о кадре из *framesParams* в параметры кадра данного объекта
Реализация:

```
framesParams = std::move(frameParams);
```

void RenderEngine::Sprite::Render (const glm::vec2 & *position*, const glm::vec2 & *size*, const float *rotation*, const float *layer* = 0, const size_t *frameId* = 0) const

Функция отрисовки объекта

Аргументы

<i>position</i>	- позиция объекта для отрисовки
-----------------	---------------------------------

<i>size</i>	- размеры объекта
<i>rotation</i>	- поворот объекта
<i>layer</i>	- слой для отрисовки на экране (по умолчанию 0)
<i>frameId</i>	- идентификатор начально кадра-текстуры (по умолчанию 0) Реализация:

```

        if (lastFrameId != frameId) //Если предыдущий кадр не равен текущему
        {
            lastFrameId = frameId; //текущий кадр становится прошлым
            const FrameParams& currentFrameDescription = framesParams[frameId];
//сохраняем параметры кадра

            const GLfloat textureCoords[] = { //сохраняем координаты текущей
текстуры
                currentFrameDescription.leftBottom.x,
currentFrameDescription.leftBottom.y,
                currentFrameDescription.leftBottom.x,
currentFrameDescription.rightTop.y,
                currentFrameDescription.rightTop.x,
currentFrameDescription.rightTop.y,
                currentFrameDescription.rightTop.x,
currentFrameDescription.leftBottom.y,
            };

            textureCoordsBuffer.UpdateBuffer(textureCoords, 2 * 4 *
sizeof(GLfloat)); //Обновляем данные объекта вершинного буфера
        }

        pShaderManager->UseShader(); //Используем шейдер

        //Вносим все необходимые изменения в объект и накладываем на него
соответствующую текстуру
        glm::mat4 model(1.f);
        model = glm::translate(model, glm::vec3(position, 0.f));
        model = glm::translate(model, glm::vec3(0.5f * size.x, 0.5f * size.y,
0.f));
        model = glm::rotate(model, glm::radians(rotation), glm::vec3(0.f, 0.f,
1.f));
        model = glm::translate(model, glm::vec3(-0.5f * size.x, -0.5f * size.y,
0.f));
        model = glm::scale(model, glm::vec3(size, 1.f));

        pShaderManager->SetMatrix4x4("modelMat", model);
        pShaderManager->SetFloat("layer", layer);
        glActiveTexture(GL_TEXTURE0);
        pTexture->Bind();

        Renderer::RenderGraphics(vertexArray, indexBuffer, *pShaderManager);
//Отрисовываем объект

```

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Renderer/Sprite.h GameEngineAndGame/src/Renderer/Sprite.cpp

Класс `RenderEngine::SpriteAnimator`

Класс, отвечающий за анимацию спрайтов, т.е смену текстуры при определенных условиях и как правило с определенной периодичностью

```
#include <SpriteAnimator.h>
```

Открытые члены

- `SpriteAnimator` (`std::shared_ptr< Sprite > pSprite`)
- `size_t GetCurrentFrame` () const
- `void Update` (const double deltaTime)

Подробное описание

Класс, отвечающий за анимацию спрайтов, т.е смену текстуры при определенных условиях и как правило с определенной периодичностью

Конструктор(ы)

`RenderEngine::SpriteAnimator::SpriteAnimator (std::shared_ptr< Sprite > pSprite)`

Конструктор класса. В нём просто присваиваются какие-либо значения полям класса

Аргументы

<i>pSprite</i>	- умный указатель на объект спрайта : <code>pSprite(std::move(pSprite))</code> , // перемещаем из переданного объекта все данные в объект поля класса <code>currentFrame(0)</code> , // Начинаем анимацию с первого кадра <code>currentFrameDuration(this->pSprite->GetFrameDuration(0))</code> , // Получаем из спрайта длительность кадра <code>currentAnimationTime(0)</code> //Начинаем отсчёт с нуля
----------------	--

Методы

`void RenderEngine::SpriteAnimator::Update (const double deltaTime)`

Метод, реализующий анимацию покадрово

Аргументы

-	deltaTime время итерации игрового цикла Реализация:
---	---

```
currentAnimationTime += deltaTime;
while (currentAnimationTime >= currentFrameDuration)
{
    currentAnimationTime -= currentFrameDuration;
    ++currentFrame;
    if (currentFrame == pSprite->GetFramesCount())
    {
        currentFrame = 0;
    }
    currentFrameDuration = pSprite->GetFrameDuration(currentFrame);
}
```

-
- **Объявления и описания членов классов находятся в файлах:**
`GameEngineAndGame/src/Renderer/SpriteAnimator.h`

- GameEngineAndGame/src/Renderer/SpriteAnimator.cpp

Благодаря выше перечисленным классам можно создать спрайт, но его ещё нужно вывести на экран. В этом поможет класс `Renderer`.

Класс `RenderEngine::Renderer`

Класс, отвечающий за отрисовку графики и управление этой отрисовкой. Также помогает создавать окно, через которое будет всё рисоваться. Принадлежит пространству имён `RenderEngine`. Вообще, всё что в этом движке отвечает за графику принадлежит пространству имен `RenderEngine`.

```
#include <Renderer.h>
```

Открытые статические члены

- static void **RenderGraphics** (const **VertexArray** &vertexArray, const **IndexBuffer** &indexBuffer, const **ShaderManager** &shaderManager)
- static void **ClearColor** (float red, float green, float blue, float alpha)
- static void **SetLayer** (const bool enabled)
- static void **Clear** ()
- static void **SetViewport** (unsigned int width, unsigned int height, unsigned int leftOffset=0, unsigned int bottomOffset=0)
- static std::string **GetRendererString** ()
- static std::string **GetVersionString** ()

Подробное описание

- Класс, отвечающий за отрисовку графики и управление этой отрисовкой. Также помогает создавать окно, через которое будет всё рисоваться. Принадлежит пространству имён `RenderEngine`. Вообще, всё что в этом движке отвечает за графику принадлежит пространству имен `RenderEngine`.

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Renderer/Renderer.h
- GameEngineAndGame/src/Renderer/Renderer.cpp

2. Компонента для работы с ресурсами

Ресурсы движка находятся в папке \res и состоят из json файла с разметкой текстур, папкой с текстурными атласами \res\textures и скриптами шейдерных программ \res\shaders. Для их считывания, обработки и передачи в нужные классы и нужен ResourceManager.

Для обработки изображений – текстурных атласов используется библиотека stb_image.h, а для работы с json – файлом, библиотека rapidjson.

Класс ResourceManager

Класс, отвечающий за загрузку ресурсов - считывание текстур с атласа текстур, разметки из json файла. Задействует все классы рендерера напрямую или косвенно.

```
#include <ResourceManager.h>
```

Открытые члены

- **ResourceManager** ()=delete
- **ResourceManager** (const **ResourceManager** &)=delete
- **ResourceManager** & operator= (const **ResourceManager** &)=delete
- **ResourceManager** & operator= (**ResourceManager** &&)=delete
- **ResourceManager** (**ResourceManager** &&)=delete

Открытые статические члены

- static void **SetExecutablePath** (const std::string &executablePath)
- static void **DestructAllRes** ()
- static std::shared_ptr< **RenderEngine::ShaderManager** > **LoadShaders** (const std::string &shaderName, const std::string &vertexPath, const std::string &fragmentPath)
- static std::shared_ptr< **RenderEngine::ShaderManager** > **GetShaderManager** (const std::string &shaderName)
- static std::shared_ptr< **RenderEngine::TextureManager** > **LoadTexture** (const std::string &textureName, const std::string &texturePath)
- static std::shared_ptr< **RenderEngine::TextureManager** > **GetTextureManager** (const std::string &textureName)
- static std::shared_ptr< **RenderEngine::Sprite** > **LoadSprite** (const std::string &spriteName, const std::string &textureName, const std::string &shaderName, const std::string &titleName="default")
- static std::shared_ptr< **RenderEngine::Sprite** > **GetSprite** (const std::string &spriteName)
- static std::shared_ptr< **RenderEngine::TextureManager** > **LoadTextureAtlas** (const std::string atlasName, const std::string texturePath, const std::vector< std::string > tilesNames, const unsigned int tileWidth, const unsigned int tileHeight)
- static bool **LoadResourcesFromJSON** (const std::string &filePath)
- static const std::vector< std::vector< std::string > > & **GetLevels** ()

Подробное описание

Класс, отвечающий за загрузку ресурсов - считывание текстур с атласа текстур, разметки из json файла. Задействует все классы рендерера напрямую или косвенно.

Конструктор(ы)

ResourceManager::ResourceManager () [delete]

Этот класс нужен нам статическим, так он должен быть одним для всего движка.

Методы

void ResourceManager::DestructAllRes () [static]

Метод, освобождающий память от всех ресурсов игры после завершения игрового цикла Реализация:

```
shaderManagers.clear();
textures.clear();
sprites.clear();
```

static const std::vector< std::vector< std::string > > & ResourceManager::GetLevels () [inline], [static]

Метод, возвращающий количество уровней в игре

std::shared_ptr< RenderEngine::ShaderManager > ResourceManager::GetShaderManager (const std::string & shaderName) [static]

Получаем шейдер с нужным именем

Аргументы

<i>shaderName</i>	- имя шейдера, который нужно получить
-------------------	---------------------------------------

Возвращает

указатель на шейдер Реализация:

```
mapShaderManager::const_iterator it = shaderManagers.find(shaderName);
if (it != shaderManagers.end())
{
    return it->second;
}
std::cerr << "Can't find the shader program :-( " << shaderName << std::endl;
return nullptr;
```

std::shared_ptr< RenderEngine::Sprite > ResourceManager::GetSprite (const std::string & spriteName) [static]

Метод, позволяющий получить спрайт с указанным именем

Аргументы

<i>spriteName</i>	- имя спрайта
-------------------	---------------

Возвращает

указатель на спрайт Реализация:

```
mapSprite::const_iterator iter = sprites.find(spriteName);
if (iter != sprites.end())
{
    return iter->second;
}
std::cerr << "Can't find the sprite: " << spriteName << std::endl;
return nullptr;
```

std::shared_ptr< RenderEngine::TextureManager >
ResourceManager::GetTextureManager (const std::string & textureName)[static]

Метод, возвращающий нужную нам текстурный атлас, который уже был загружен

Аргументы

<i>textureName</i>	- имя текстурного атласа Реализация:
--------------------	--------------------------------------

```
mapTextureManager::const_iterator it = textures.find(textureName);
if (it != textures.end())
{
    return it->second;
}
std::cerr << "Can't find the texture: " << textureName << std::endl;
return nullptr;
```

bool ResourceManager::LoadResourcesFromJSON (const std::string & filePath)[static]

Метод, сопоставляющий разметку ресурсов из файла json с ресурсами в файлах и загружающий все ресурсы

Аргументы

<i>filePath</i>	- путь к json - файлу с разметкой
-----------------	-----------------------------------

Возвращает

загрузились ли ресурсы с разметкой или нет Реализация:

```
const std::string JSONString = GetTextFromFile(filePath); //Считываем весь
текст с json-файла
if (JSONString.empty())
{
    std::cerr << "File with JSON is empty :-( " << std::endl;
    return false;
}

rapidjson::Document doc;
rapidjson::ParseResult parseResult = doc.Parse(JSONString.c_str()); //парсим
текст в премлемый формат для работы с json
if (!parseResult)
{
    std::cerr << "JSON parse error :-( " <<
rapidjson::GetParseError_En(parseResult.Code()) << " In file " << filePath <<
std::endl;
    return false;
}

auto shadersIter = doc.FindMember("shaders"); //находим заголовок шейдеров

if (shadersIter != doc.MemberEnd()) //если такой заголовок есть и под ним что-
то написано, загружаем шейдеры
{
    for (const auto &shader : shadersIter->value.GetArray())
    {
        const std::string name = shader["name"].GetString();
        const std::string vertexPath = shader["vertex_path"].GetString();
        const std::string fragmentPath = shader["fragment_path"].GetString();
        LoadShaders(name, vertexPath, fragmentPath);
    }
}

auto textureAtlasIter = doc.FindMember("TextureAtlases");

if (textureAtlasIter != doc.MemberEnd()) //аналогично загружаем все данные со
всех атласов текстур
{
    for (const auto& atlas : textureAtlasIter->value.GetArray())
    {
        const std::string name = atlas["name"].GetString();
        const std::string atlasPath = atlas["filePath"].GetString();
        const unsigned int tileWidth = atlas["tileWidth"].GetUint();
```

```

        const unsigned int tileHeight = atlas["tileHeight"].GetUInt();

        const auto tilesArray = atlas["tiles"].GetArray();
        std::vector<std::string> tiles;
        tiles.reserve(tilesArray.Size());
        for (const auto& tile : tilesArray)
        {
            tiles.emplace_back(tile.GetString());
        }
        LoadTextureAtlas(name, atlasPath, std::move(tiles), tileWidth,
tileHeight);
    }
}
auto spritesIter = doc.FindMember("sprites"); //Среди текстур обозначаем
спрайты, привязываем к ним шейдеры. Если у них есть анимация, загружаем время
анимаций и располагаем их по порядку.
if (spritesIter != doc.MemberEnd())
{
    for (const auto& currentSprite : spritesIter->value.GetArray())
    {
        const std::string name = currentSprite["name"].GetString();
        const std::string textureAtlas =
currentSprite["textureAtlas"].GetString();
        const std::string shader = currentSprite["shader"].GetString();
        const std::string tile = currentSprite["initialTile"].GetString();

        auto pSprite = LoadSprite(name, textureAtlas, shader, tile);
        if (!pSprite)
        {
            continue;
        }
        auto framesIter = currentSprite.FindMember("frames");
        if (framesIter != currentSprite.MemberEnd())
        {
            const auto framesArray = framesIter->value.GetArray();
            std::vector<RenderEngine::Sprite::FrameParams> framesParams;
            framesParams.reserve(framesArray.Size());
            for (const auto& frame : framesArray)
            {
                const std::string tile = frame["tile"].GetString();
                const double duration = frame["duration"].GetDouble();
                const auto pTextureAtlas = GetTextureManager(textureAtlas);
                const auto pTile = pTextureAtlas->GetTile(tile);
                framesParams.emplace_back(pTile.leftBottom, pTile.rightTop,
duration);
            }
            pSprite->InsertFrames(std::move(framesParams));
        }
    }
}

auto levelsIt = doc.FindMember("levels"); //считываем разметку каждого
уровня
if (levelsIt != doc.MemberEnd())
{
    for (const auto& currentLevel : levelsIt->value.GetArray())
    {
        const auto description = currentLevel["description"].GetArray();
        std::vector<std::string> levelRows;
        levelRows.reserve(description.Size());
        size_t maxLength = 0;
        for (const auto& currentRow : description)
        {
            levelRows.emplace_back(currentRow.GetString());
            if (maxLength < levelRows.back().length())
            {
                maxLength = levelRows.back().length();
            }
        }
        levels.emplace_back(std::move(levelRows));
    }
}
return true;

```

std::shared_ptr< RenderEngine::ShaderManager > ResourceManager::LoadShaders (const std::string & *shaderName*, const std::string & *vertexPath*, const std::string & *fragmentPath*)[static]

Загружает шейдер с указанным именем и возвращает умный указатель на него

Аргументы

<i>shaderName</i>	- имя шейдера
<i>vertexPath</i>	- путь к файлу с исходным кодом программы шейдера вершин, написанной на GLSL
<i>fragmentPath</i>	- путь к файлу с исходным кодом программы шейдера фрагментов, написанной на GLSL

Возвращает

указатель на загруженный шейдер Реализация:

```
std::string vertexString = GetTextFromFile(vertexPath); // считываем текст
вертексной программы
if (vertexString.empty()) //Если что-то пошло не так, выводим ошибку
{
    std::cerr << "Can't find vertex shader :-( " << std::endl;
    return nullptr;
}
//Аналогично поступаем с шейдерной программой
std::string fragmentString = GetTextFromFile(fragmentPath);
if (fragmentString.empty())
{
    std::cerr << "Can't find fragment shader :-( " << std::endl;
    return nullptr;
}

std::shared_ptr<RenderEngine::ShaderManager>& newShader =
shaderManagers.emplace(shaderName,
std::make_shared<RenderEngine::ShaderManager>(vertexString,
fragmentString)).first->second; //Создаём новый шейдер из вертексной
программы и фрагментной
if (newShader->IsCompiled()) //Если шейдер собрался удачно, возвращаем его,
иначе выводим ошибку и возвращаем nullptr
{
    return newShader;
}

std::cerr << "Can't load shader program:" << std::endl << "Vertex program: "
<< vertexPath << std::endl << "Fragment program: " << fragmentPath <<
std::endl;
return nullptr;
```

std::shared_ptr< RenderEngine::Sprite > ResourceManager::LoadSprite (const std::string & *spriteName*, const std::string & *textureName*, const std::string & *shaderName*, const std::string & *titleName* = "default")[static]

Загружает нужный спрайт и возвращает указатель на него

Аргументы

--	--

param textureName - имя атласа текстур, с которого будет получен спрайт

Аргументы

<i>shaderName</i>	- имя шейдера, который соответствует данному спрайту
<i>titleName</i>	- имя нужного нам спрайта (по умолчанию имя default), с которым объект изначально будет отрисовываться

Возвращает

- указатель на спрайт Реализация:

```
auto pTexture = GetTextureManager(textureName);
if (!pTexture)
{
    std::cerr << "Can't find the texture: " << textureName << " for the
sprite: " << spriteName << std::endl;
}

auto pShader = GetShaderManager(shaderName);
if (!pShader)
{
    std::cerr << "Can't find the shader: " << shaderName << " for the sprite:
" << spriteName << std::endl;
}

std::shared_ptr<RenderEngine::Sprite> newSprite = sprites.emplace(spriteName,
std::make_shared<RenderEngine::Sprite>(pTexture,

titleName,

pShader)).first->second;

return newSprite;
```

**std::shared_ptr< RenderEngine::TextureManager > ResourceManager::LoadTexture
(const std::string & textureName, const std::string & texturePath)[static]**

Метод, который загружает и возвращает текстурный атлас

Аргументы

<i>textureName</i>	- имя текстурного атласа
<i>texturePath</i>	- путь к текстурному атласу

Возвращает

указатель на текстурный атлас На наше счастье библиотека загрузки изображений **stb_image.h**, реализованная в один заголовочный файл, поддерживает загрузку RGBE файлов, возвращая массив чисел с плавающей точкой – то что надо для наших целей! Добавив библиотеку в свой проект загрузка данных изображений реализуется предельно просто: Реализация:

```
int channels = 0; //количество цветовых каналов
int width = 0; // ширина текстуры
int height = 0; //высота текстуры

stbi set flip vertically on load(true);
unsigned char* pixels = stbi_load(std::string(path + "/" +
texturePath).c_str(), &width, &height, &channels, 0);

if (!pixels)
{
    std::cerr << "Can't load image: " << texturePath << std::endl;
    return nullptr;
}

std::shared_ptr<RenderEngine::TextureManager> newTexture =
textures.emplace(textureName,
std::make_shared<RenderEngine::TextureManager>(width, height, pixels,
channels, GL_NEAREST, GL_CLAMP_TO_EDGE)).first->second;
stbi_image_free(pixels);
return newTexture;
```

**std::shared_ptr< RenderEngine::TextureManager >
ResourceManager::LoadTextureAtlas (const std::string atlasName, const std::string**

***texturePath*, const std::vector< std::string > *tilesNames*, const unsigned int *tileWidth*, const unsigned int *tileHeight*)[static]**

Метод, считывающий все текстуры с текстурного атласа

Аргументы

<i>atlasName</i>	- имя атласа
<i>texturePath</i>	- расположение атласа текстур
<i>tilesNames</i>	- имена текстур
<i>tileWidth</i>	- ширина спрайта
<i>tileHeight</i>	- высота спрайта

Возвращает

указатель на текстуру Реализация:

```
const unsigned int tileHeight) {
    auto pTexture = LoadTexture(std::move(atlasName), std::move(texturePath));
    if (pTexture)
    {
        const unsigned int textureWidth = pTexture->GetWidth();
        const unsigned int textureHeight = pTexture->GetHeight();
        unsigned int currentTextureOffsetX = 0;
        unsigned int currentTextureOffsetY = textureHeight;
        for (auto& currentTileName : tilesNames)
        {
            glm::vec2 leftBottom(static_cast<float>(currentTextureOffsetX + 0.01f)
            / textureWidth, static_cast<float>(currentTextureOffsetY - tileHeight + 0.01f)
            / textureHeight);
            glm::vec2 rightTop(static_cast<float>(currentTextureOffsetX +
            tileWidth - 0.01f) / textureWidth, static_cast<float>(currentTextureOffsetY -
            0.01f) / textureHeight);

            pTexture->AddTile(std::move(currentTileName), leftBottom, rightTop);

            currentTextureOffsetX += tileWidth;
            if (currentTextureOffsetX >= textureWidth)
            {
                currentTextureOffsetX = 0;
                currentTextureOffsetY -= tileHeight;
            }
        }
    }
    return pTexture;
}
```

void ResourceManager::SetExecutablePath (const std::string & executablePath)[static]

Метод устанавливающий в менеджер ресурсов путь к папке со всеми ресурсами игры. Так как exe лужит в одной папке с папкой res, то передать в executablePath следует argv[0]

Аргументы

<i>executablePath</i>	- путь к папке с res Реализация:
-----------------------	----------------------------------

```
size_t found = executablePath.find_last_of("/\\");
path = executablePath.substr(0, found);
```

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Resources/ResourceManager.h
- GameEngineAndGame/src/Resources/ResourceManager.cpp

3. Физическая компонента

PhysicsEngineManager – пространство имён, отвечающее за физику в движке.

Класс **PhysicsEngineManager::PhysicsEngine**

Класс, отвечающий за взаимодействия игровых объектов с игроком и друг с другом. Статический

```
#include <PhysicsEngine.h>
```

Открытые члены

- **PhysicsEngine** (const **PhysicsEngine** &)=delete
- **PhysicsEngine** & operator= (const **PhysicsEngine** &)=delete
- **PhysicsEngine** & operator= (**PhysicsEngine** &&)=delete
- **PhysicsEngine** (**PhysicsEngine** &&)=delete

Открытые статические члены

- static void **Update** (const double delta)
- static void **AddDynamicGameObject** (std::shared_ptr< **GameObjectInterface** > pGameObject)
- static void **DeleteAll** ()
- static void **SetCurrentLevel** (std::shared_ptr< **Level** > pLevel)

Подробное описание

Класс, отвечающий за взаимодействия игровых объектов с игроком и друг с другом. Статический

Методы

void PhysicsEngineManager::PhysicsEngine::AddDynamicGameObject (std::shared_ptr< GameObjectInterface > pGameObject) [static]

Добавляет в сет указателей на объекты, способные взаимодействовать, pGameObject - ещё один объект

Аргументы

<i>pGameObject</i>	- умный указатель на объект, который надо добавить Реализация:
--------------------	--

```
dynamicObjects.insert(std::move(pGameObject));
```

void PhysicsEngineManager::PhysicsEngine::DeleteAll () [static]

Очищает данные об уровне в игре и игровых объектах Реализация:

```
dynamicObjects.clear();  
pCurrentLevel.reset();
```

void PhysicsEngineManager::PhysicsEngine::SetCurrentLevel (std::shared_ptr< Level > pLevel) [static]

Метод, передающий pCurrentLevel текущий уровень, а именно обменивается данными с pLevel

Аргументы

<i>pLevel</i>	- указатель на уровень Реализация:
---------------	------------------------------------

```
pCurrentLevel.swap(pLevel);
```

void PhysicsEngineManager::PhysicsEngine::Update (const double *delta*)[static]

Метод, обрабатывающий каждый кадр во время игры и проверяющий, есть ли взаимодействия объектов. Если есть, выполняется их взаимодействие

Аргументы

<i>delta</i>	- единица времени кадра Реализация:
--------------	-------------------------------------

```
for (auto& currentObject : dynamicObjects)
{
    if (currentObject->GetCurrentVelocity() > 0)
    {
        if (currentObject->GetCurrentDirection().x != 0.f)
        {
            currentObject->GetCurrentPosition() = glm::vec2(currentObject->GetCurrentPosition().x,
static cast<unsigned int>(currentObject->GetCurrentPosition().y / 4.f + 0.5f) * 4.f);
        }
        else if (currentObject->GetCurrentDirection().y != 0.f)
        {
            currentObject->GetCurrentPosition() = glm::vec2(static_cast<unsigned
int>(currentObject->GetCurrentPosition().x / 4.f + 0.5f) * 4.f, currentObject-
>GetCurrentPosition().y);
        }
        const auto newPosition = currentObject->GetCurrentPosition() + currentObject-
>GetCurrentDirection() * static_cast<float>(currentObject->GetCurrentVelocity() * delta);
        const auto& colliders = currentObject->GetColliders();
        std::vector<std::shared_ptr<GameObjectInterface>> objectsToCheck = pCurrentLevel-
>GetLevelObjects(newPosition, newPosition + currentObject->GetSize());

        bool hasCollision = false;
        for (const auto& currentObjectToCheck : objectsToCheck)
        {
            const auto& collidersToCheck = currentObjectToCheck->GetColliders();
            if (currentObjectToCheck->IsCollides(currentObject->GetObjectType()) &&
!collidersToCheck.empty())
            {
                if (HasInteraction(colliders, newPosition, collidersToCheck, currentObjectToCheck-
>GetCurrentPosition()))
                {
                    hasCollision = true;
                    currentObjectToCheck->OnCollision();
                    break;
                }
            }
        }

        if (!hasCollision)
        {
            currentObject->GetCurrentPosition() = newPosition;
        }
        else
        {
            if (currentObject->GetCurrentDirection().x != 0.f)
            {
                currentObject->GetCurrentPosition() = glm::vec2(static_cast<unsigned
int>(currentObject->GetCurrentPosition().x / 8.f + 0.5f) * 8.f, currentObject-
>GetCurrentPosition().y);
            }
            else if (currentObject->GetCurrentDirection().y != 0.f)
            {
                currentObject->GetCurrentPosition() = glm::vec2(currentObject-
>GetCurrentPosition().x, static_cast<unsigned int>(currentObject->GetCurrentPosition().y / 8.f +
0.5f) * 8.f);
            }
            currentObject->OnCollision();
        }
    }
}
```

Объявления и описания членов классов находятся в файлах:

- `GameEngineAndGame/src/PhysicsEngine/PhysicsEngine.h`
- `GameEngineAndGame/src/PhysicsEngine/PhysicsEngine.cpp`

4. Компонента программного интерфейса

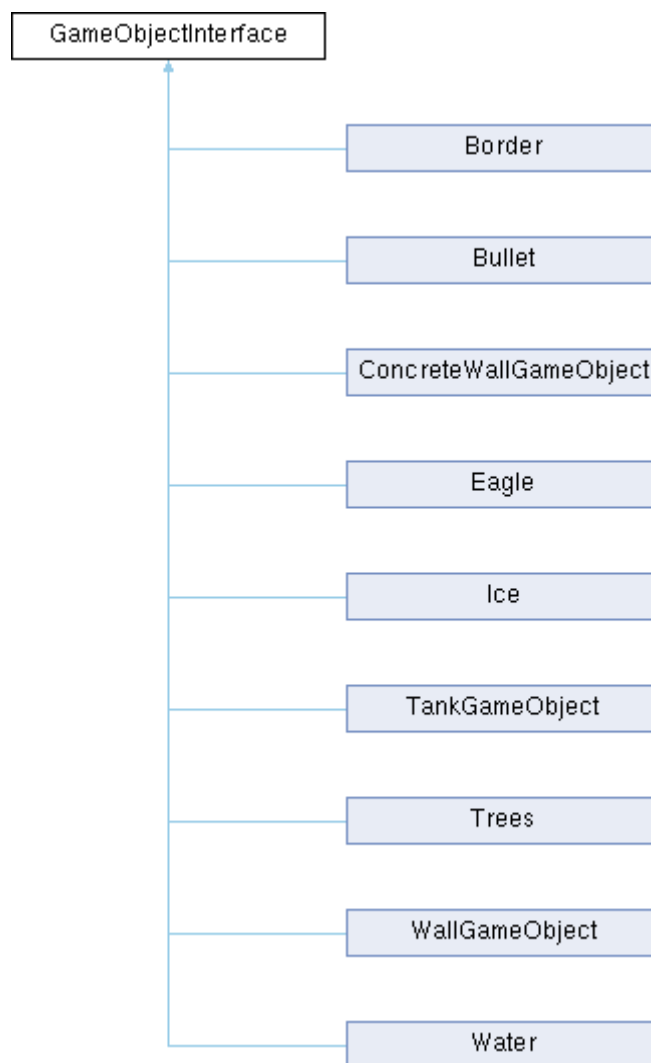
В классах **GameObjectInterface** и **LevelInterface** собраны все необходимые функции, которые разработчику необходимо применить в создании игры на данном движке. Также здесь собраны абстрактные методы, присущие каждому игровому объекту и уровню соответственно.

Класс GameObjectInterface

Виртуальный класс, содержащий в себе поля и методы, общие для всех возможных игровых объектов. Является связующим звеном между объектом и физическим движком. Наследование от этого класса при создании игрового объекта обязательно!

```
#include <GameObjectInterface.h>
```

Граф наследования:GameObjectInterface:



Все игровые объекты должны быть наследниками этого класса, как в этом примере

Открытые типы

- enum class **ObjectType** { **Constant_Satic_Object**, **Constant_Dynamic_Object**, **Temporary_Satic_Object**, **Temporary_Dynamic_Object** }

Класс, хранящий перечисления типов игровых объектов (Подвижный, способный к взаимодействию объект, подвижный неспособный к взаимодействию объект, неподвижный, способный к взаимодействию объект, неподвижный неспособный к взаимодействию объект)

Открытые члены

- **GameObjectInterface** (const **ObjectType** objectType, const glm::vec2 &position, const glm::vec2 &size, const float rotation, const float layer)
- virtual void **Render** () const =0
- virtual void **UpdateFrame** (const double delta)
- virtual glm::vec2 & **GetCurrentPosition** ()
- virtual glm::vec2 & **GetCurrentDirection** ()
- virtual double **GetCurrentVelocity** ()
- virtual void **SetVelocity** (const double velocity)
- const std::vector< **PhysicsEngineManager::BoxCollider** > & **GetColliders** () const
- const glm::vec2 & **GetSize** () const
- **ObjectType** **GetObjectType** () const
- virtual bool **IsCollides** (const **ObjectType** objectType)
- virtual void **OnCollision** ()

Защищенные данные

- glm::vec2 **GOIposition**
Позиция объекта
- glm::vec2 **GOIsize**
Размер объекта
- glm::vec2 **GOIdirection**
Поворот объекта
- **ObjectType** **GOIobjectType**
Тип объекта
- double **GOIvelocity**
Скорость объекта
- float **GOIrotation**
Поворот объекта
- float **GOIplayer**
Слой объекта
- std::vector< **PhysicsEngineManager::BoxCollider** > **boxColliders**
Коллайдеры объекта

Подробное описание

Виртуальный класс, содержащий в себе поля и методы, общие для всех возможных игровых объектов. Является связующим звеном между объектом и физическим движком. Наследование от этого класса при создании игрового объекта обязательно!

Конструктор(ы)

GameObjectInterface::GameObjectInterface (const ObjectType objectType, const glm::vec2 & position, const glm::vec2 & size, const float rotation, const float layer)

Конструктор класса примет следующие параметры:

Аргументы

<i>objectType</i>	- тип объекта
<i>position</i>	- позиция объекта
<i>size</i>	- размер объекта
<i>rotation</i>	- поворот объекта
<i>layer</i>	- слой объекта Больше в конструкторе ничего не происходит

Методы

const std::vector< PhysicsEngineManager::BoxCollider > & GameObjectInterface::GetColliders ()
const [inline]

Возвращает коллайдеры объекта

glm::vec2 & GameObjectInterface::GetCurrentDirection () [virtual]

Возвращает текущий поворот объекта

glm::vec2 & GameObjectInterface::GetCurrentPosition () [virtual]

Возвращает текущую позицию объекта

double GameObjectInterface::GetCurrentVelocity () [virtual]

Возвращает скорость объекта

ObjectType GameObjectInterface::GetObjectType () const [inline]

Возвращает тип объекта

const glm::vec2 & GameObjectInterface::GetSize () const [inline]

Возвращает координаты границ объекта - т.е его размер

virtual bool GameObjectInterface::IsCollides (const ObjectType objectType) [inline], [virtual]

Функция, возвращающая значение, относительно типа объекта, может ли объект взаимодействовать с другими

Переопределяется в **Water** (*cmp.Error! Bookmark not defined.*).

virtual void GameObjectInterface::OnCollision () [inline], [virtual]

Возвращает булево значение, находится ли объект во взаимодействии сейчас или нет

Переопределяется в **Bullet** (*cmp.Error! Bookmark not defined.*).

virtual void GameObjectInterface::Render () const [pure virtual]

Функция, отвечающая за отрисовку объекта, которую разработчик должен будет переопределить

Замещается в **Border** (*cmp.Error! Bookmark not defined.*), **Bullet** (*cmp.Error! Bookmark not defined.*), **ConcreteWallGameObject** (*cmp.Error! Bookmark not defined.*), **Eagle** (*cmp.Error! Bookmark not defined.*), **Ice** (*cmp.Error! Bookmark not defined.*), **TankGameObject** (*cmp.Error! Bookmark not defined.*), **Trees** (*cmp.Error! Bookmark not defined.*), **WallGameObject** (*cmp.Error! Bookmark not defined.*) и **Water** (*cmp.Error! Bookmark not defined.*).

void GameObjectInterface::SetVelocity (const double velocity)[virtual]

Устанавливает на объекте новую скорость

Переопределяется в **TankGameObject** (*cmp.Error! Bookmark not defined.*).

virtual void GameObjectInterface::UpdateFrame (const double delta)[inline], [virtual]

Функция, обработку физики объекта каждый кадр, которую разработчик должен будет переопределить

Переопределяется в **ConcreteWallGameObject** (*cmp.Error! Bookmark not defined.*), **Eagle** (*cmp.Error! Bookmark not defined.*), **TankGameObject** (*cmp.Error! Bookmark not defined.*), **WallGameObject** (*cmp.Error! Bookmark not defined.*) и **Water** (*cmp.Error! Bookmark not defined.*).

Объявления и описания членов классов находятся в файлах:

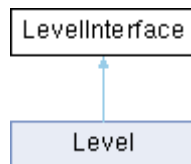
- GameEngineAndGame/src/Game/GameObjects/GameObjectInterface.h
- GameEngineAndGame/src/Game/GameObjects/GameObjectInterface.cpp

Класс LevelInterface

Виртуальный класс, содержащий в себе поля и методы, общие для всех создаваемых уровней. Наследование от этого класса при создании уровня обязательно!

```
#include <LevelInterface.h>
```

Граф наследования:LevelInterface:



Все уровни, должны быть наследниками этого класса, как этом примере

Открытые члены

- virtual void **Render** () const
- virtual void **Update** (const double delta)
- virtual size_t **GetLevelWidth** () const
- virtual size_t **GetLevelHeight** () const
- virtual std::vector< std::shared_ptr< **GameObjectInterface** > > **GetLevelObjects** (const glm::vec2 &bottomLeft, const glm::vec2 &topRight) const

Статические открытые данные

- static constexpr unsigned int **BLOCK_SIZE** = 16
Размер в пикселях единицы уровня (например блок бетонной стены имеет данный размер)

Защищенные данные

- size_t **width** = 0
- size_t **height** = 0
- unsigned int **widthPixels** = 0
- unsigned int **heightPixels** = 0
- std::vector< std::shared_ptr< **GameObjectInterface** > > **levelObjects**

Подробное описание

Виртуальный класс, содержащий в себе поля и методы, общие для всех создаваемых уровней. Наследование от этого класса при создании уровня обязательно!

Методы

size_t LevelInterface::GetLevelHeight () const [virtual]

Возвращает высоту уровня

std::vector< std::shared_ptr< GameObjectInterface > > LevelInterface::GetLevelObjects (const glm::vec2 & bottomLeft, const glm::vec2 & topRight) const [virtual]

Возвращает объекты, находящиеся рядом с игроком. Метод нужен для получения тех не многих объектов, с которыми рядом находится объект, относительно которого требуется найти объекты. Чтобы обработать предполагаемые взаимодействия этих объектов

Аргументы

<i>bottomLeft</i>	- координаты левого нижнего угла игрового объекта, относительно которого ищется объекты рядом.
<i>topRight</i>	- координаты правого верхнего угла игрового объекта, относительно которого ищется объекты рядом.

Возвращает

вектор указателей на игровые объекты, с которыми находится рядом объект, относительно которого требуется найти объекты поблизости Реализация:

```
std::vector<std::shared_ptr<GameObjectInterface>> returnLevelObjects;
returnLevelObjects.reserve(9);

glm::vec2 newBottomLeft(std::clamp(bottomLeft.x - BLOCK_SIZE, 0.f,
static_cast<float>(widthPixels)), std::clamp(heightPixels - bottomLeft.y + BLOCK_SIZE / 2, 0.f,
static_cast<float>(heightPixels)));
glm::vec2 newTopRight(std::clamp(topRight.x - BLOCK_SIZE, 0.f,
static_cast<float>(widthPixels)), std::clamp(heightPixels - topRight.y + BLOCK_SIZE / 2, 0.f,
static_cast<float>(heightPixels)));

size_t startX = static_cast<size_t>(floor(newBottomLeft.x / BLOCK_SIZE));
size_t endX = static_cast<size_t>(ceil(newTopRight.x / BLOCK_SIZE));

size_t startY = static_cast<size_t>(floor(newTopRight.y / BLOCK_SIZE));
size_t endY = static_cast<size_t>(ceil(newBottomLeft.y / BLOCK_SIZE));

for (size_t currentColumn = startX; currentColumn < endX; currentColumn++)
{
    for (size_t currentRow = startY; currentRow < endY; currentRow++)
    {
        auto& currentObject = levelObjects[currentRow * width + currentColumn];
        if (currentObject)
        {
            returnLevelObjects.push_back(currentObject);
        }
    }
}

if (endX >= width)
{
    returnLevelObjects.push_back(levelObjects[levelObjects.size() - 1]);
}
if (startX <= 1)
{
    returnLevelObjects.push_back(levelObjects[levelObjects.size() - 2]);
}
if (startY <= 1)
{
    returnLevelObjects.push_back(levelObjects[levelObjects.size() - 3]);
}
if (endY >= height)
{
    returnLevelObjects.push_back(levelObjects[levelObjects.size() - 4]);
}

return returnLevelObjects;
```

size_t LevelInterface::GetLevelWidth () const [virtual]

Возвращает ширину уровня

void LevelInterface::Render () const [virtual]

Функция, отвечающая за отрисовку уровня Реализация:

```
for (const auto & currentLevelObject : levelObjects)
{
    if (currentLevelObject)
    {
        currentLevelObject->Render();
    }
}
```

void LevelInterface::Update (const double *delta*)[virtual]

Метод, отвечающий за обновление данных об объектах на текущем уровне за единицу времени кадра

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/Game/LevelInterface.h
- GameEngineAndGame/src/Game/LevelInterface.cpp

Класс Timer

Класс, необходимый для событий/анимаций в игре, которые не зациклены и не длятся бесконечно. Чтобы их можно было остановить

```
#include <Timer.h>
```

Открытые члены

- void **Update** (const double delta)
- void **Start** (const double duration)
- void **SetCallback** (std::function< void()> callback)

Подробное описание

Класс, необходимый для событий/анимаций в игре, которые не зациклены и не длятся бесконечно. Чтобы их можно было остановить

Методы

void Timer::SetCallback (std::function< void()> callback)

Получает и устанавливает в таймере функцию обратного отклика

void Timer::Start (const double duration)

Функция, запускающая таймер

Аргументы

<i>duration</i>	- длительность таймера
-----------------	------------------------

```
timeLeft = duration;  
isRunning = true;
```

void Timer::Update (const double delta)

Функция проверяющий, истёкло ли время таймера. Включает

Аргументы

<i>delta</i>	-единица времени кадра Реализация:
--------------	------------------------------------

```
if (isRunning)  
{  
    timeLeft -= delta;  
    if (timeLeft <= 0)  
    {  
        isRunning = false;  
        callback();  
    }  
}
```

Объявления и описания членов классов находятся в файлах:

- GameEngineAndGame/src/EngineSystem/Timer.h
- GameEngineAndGame/src/EngineSystem/Timer.cpp

5. Игровой цикл

```
auto lastTime = std::chrono::high_resolution_clock::now(); ///  
//<Переменная времени, необходимая для вычисления длительности кадра  
  
while (!glfwWindowShouldClose(pWindow))  
{  
    auto currentTime = std::chrono::high_resolution_clock::now(); ///  
//<Переменная времени, необходимая для вычисления длительности кадра  
    double timeDuration = std::chrono::duration<double, std::milli>(currentTime -  
lastTime).count(); ///  
//<Длительность кадра  
    lastTime = currentTime;  
    game->Update(timeDuration); ///  
//<Обновление игровых данных  
    PhysicsEngineManager::PhysicsEngine::Update(timeDuration); ///  
//<Обновление физических данных  
    RenderEngine::Renderer::Clear(); ///  
//<Очистка прошлого кадра  
  
    game->Render(); ///  
//<Отрисовка нового кадра  
    glfwSwapBuffers(pWindow);  
  
    glfwPollEvents();  
}  
game = nullptr;  
ResourceManager::DestructAllRes();  
}  
//< Очистка ресурсов
```

Кроме того, в данном классе происходит много других важных вещей, таких как:

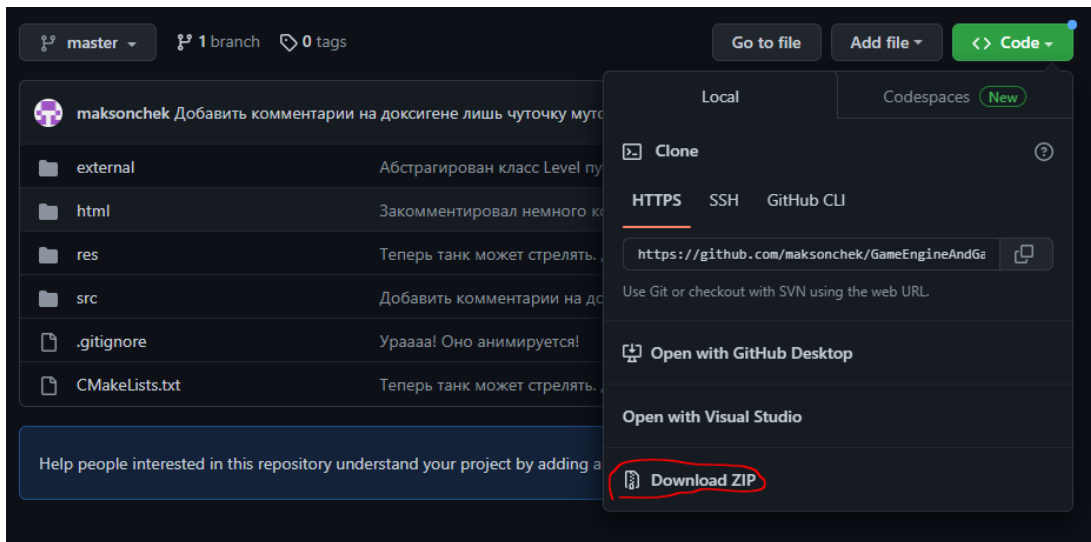
- Создание экземпляра класса игры
- Функция, отрисовывающая игру в нормальном для восприятия виде при изменении размеров игрового окна
- Функция, принимающая нажатия кнопок и передающая их классу игры для дальнейшей обработки
- Инициализация glfw
- Создание окошка для вывода картинок
- Установка пути к папке с ресурсами игры
- Установка размеров окна

Инструкция по установке

! Движок использует стандарт c++ 17

1) Перейти в [github проекта](#)

2) Скачать архив проекта или клонировать репозиторий или использовать любой другой способ получения файлов проекта



3) создать папку build

build	17.01.2023 1:29	Папка с файлами	
external	12.01.2023 20:57	Папка с файлами	
html	12.01.2023 20:57	Папка с файлами	
res	12.01.2023 20:57	Папка с файлами	
src	12.01.2023 20:57	Папка с файлами	
.gitignore	12.01.2023 20:57	Текстовый докум...	1 КБ
CMakeLists.txt	12.01.2023 20:57	Текстовый докум...	3 КБ

4) Зайти в неё и открыть консоль/gitbash/что-то для взаимодействия с cmake.

Ввести по очереди команды

cmake ..

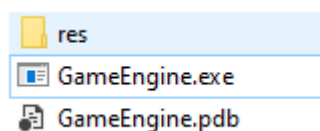
cmake --build .

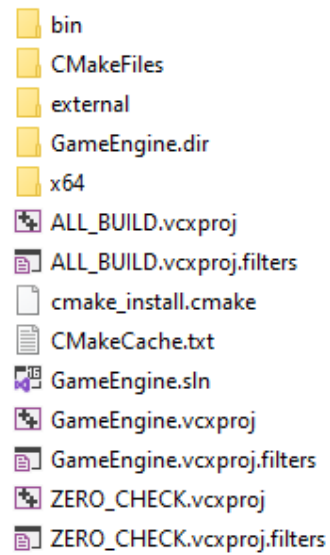
5) После этого в папке build появятся следующие файлы:

GameEngine.sln – проект движка, код которого теперь можно увидеть.

(Не забывайте обновить стандарт c++ до 17)

При успешной сборке в папке \build\bin\Debug появится exe файл

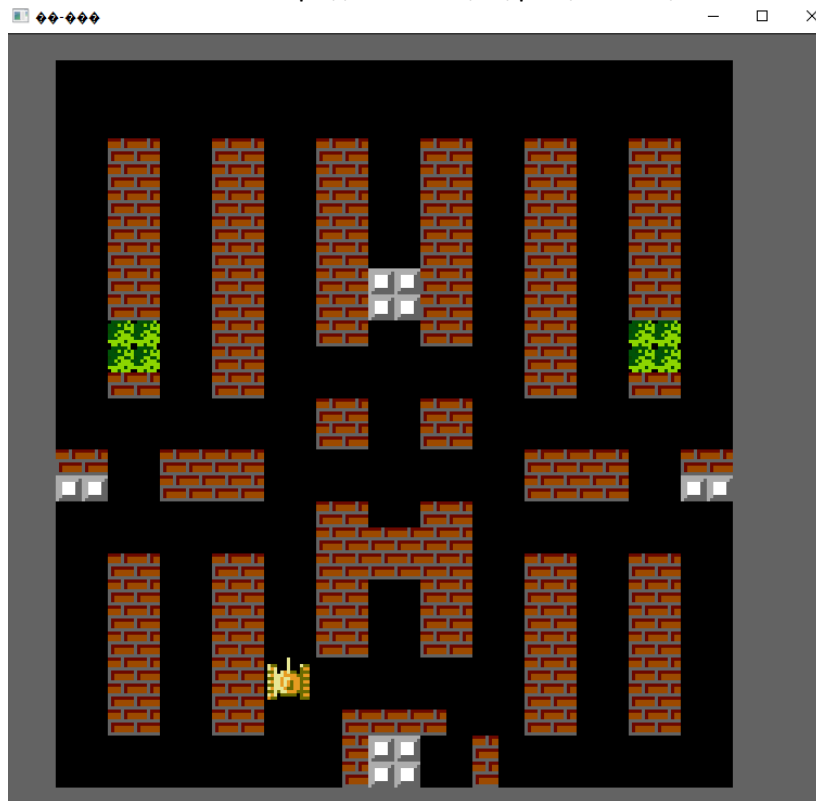




Файлы при успешном билде

6) Открываем exe для проверки успешности сборки

<https://habr.com/ru/post/142126/>



Инструкция по использованию движка

Легче всего научиться пользоваться движком – создать игру. Но сначала нужно открыть сам движок. Для этого нужно запустить GameEngine.sln в папке build и назначить GameEngine запускаемым проектом.

Теперь нужно спроектировать создаваемую игру. Для демонстрации возможностей движка.

Чтобы показать, на что способен данный движок отлично подойдёт игра “Танчики”, которая довольно стара и имеет в интернете все ресурсы, необходимые для её создания.

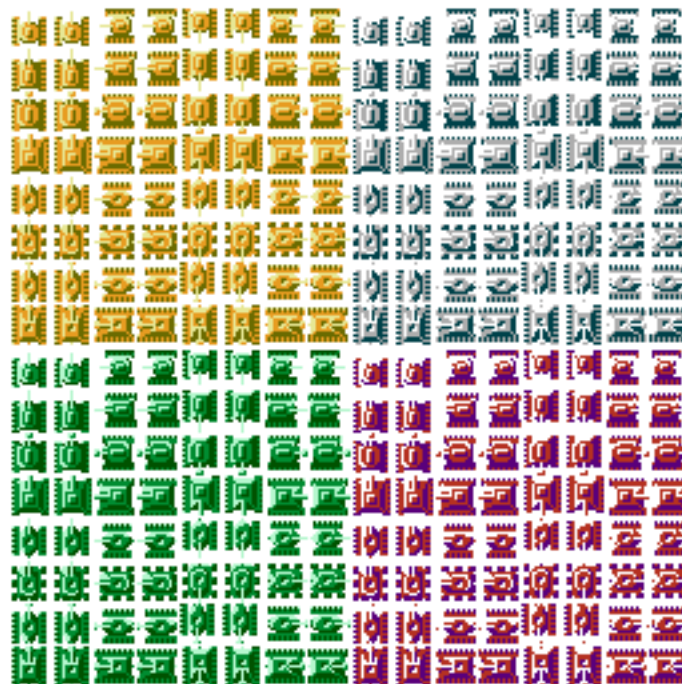
1) Подготовка ресурсов



Кадры для анимации событий. Нам понадобится только первая строчка



Текстуры игровых объектов



Виды танка и кадры для их анимации. Нам понадобится только первая строчка

Эти картинки и есть **текстурные атласы**. Их нужно поместить в папку \res\textures. Иначе движок не сможет найти эти ресурсы.

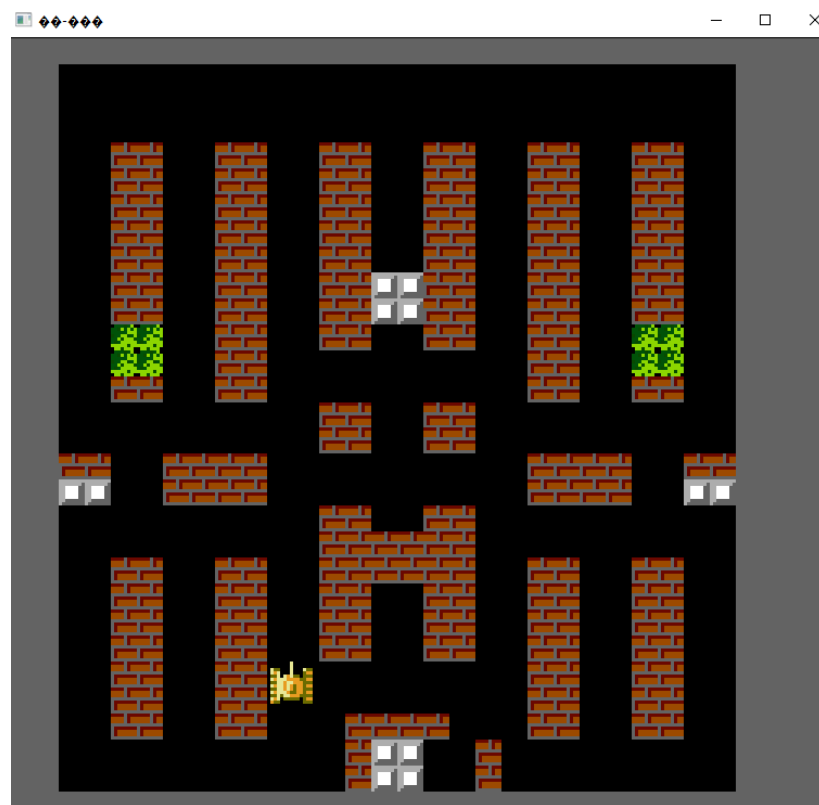
Теперь необходимо закодировать эти картинки. Так чтобы программе было понятно, с каким из кусков этого атласа она работает.

Уровень можно размечать [следующим способом](#):

	0		1		2		3		4
	5		6		7		8		9
	A		B		C		D		E
	F		G		H		I		J

Каждая буква отвечает за отдельную текстуру.

Чтобы спроектировать такой уровень,



Вид уровня

В файле `\res\gameResources.json` нужно ввести следующий текст:

```
"levels": [
{
  "description": [
    "MDDDDDNDDDDO",
    "D1D1D1D1D1D1D",
    "D4D4D4D4D4D4D",
    "D4D4D4D4D4D4D",
    "D4D4D494D4D4D",
    "DBD4D3D3D4DBD",
    "D3D3D1D1D3D3D",
```

```

        "1D11D3D3D11D1",
        "8D33D1D1D33D8",
        "D1D1D444D1D1D",
        "D4D4D4D4D4D4D",
        "D4D4D3D3D4D4D",
        "D4D4DH1GD4D4D",
        "DDDDK09s2LDDDD"
    ]
}
]

```

Прежде чем идти дальше, стоит понять структуру `\res\gameResources.json`.

!Требование: в текстурных атласах все текстуры должны быть расположены вплотную. Так как движок считывает их прямоугольниками. Например размером 8 на 8 пикселей.

Ниже приведена структура файла и примеры тех возможностей, которые можно реализовать. Сама структура должна соблюдаться вплоть до имен json-переменных, иначе движок не заработает.

```

{
    "shaders": [ //Здесь указывается путь к шейдерам и задаются их имена. При желании разработчик
        может добавить свои. В движке предусмотрен один базовый шейдер
        {
            "name": "SpriteShader", //Имя шейдера
            "vertex_path": "res/shaders/vertexSprite.txt", //Путь к вершинному шейдеру
            "fragment_path": "res/shaders/fragmentSprite.txt" //Путь к фрагментному шейдеру
        }
    ],

    "TextureAtlases": [ //Здесь размечаются все объекты на текстурных атласах, которые нам
        нужны. По порядку, в котором они расположены в атласах!
        {
            "name": "MiniMapTextureAtlas", //Имя атласа. Можно написать любое. Оно ещё понадобится
            "filePath": "res/textures/miniAssets.png", //Путь к атласу
            "tileWidth": 8, //Размеры одной текстуры
            "tileHeight": 8,
            "tiles": [
                //Здесь идёт перечисление всех текстур. Мы просто задаём им имена.
                "break",
                "concrete",
                "brickWall_All",
                "water1",
                "water2",
                "water3"
            ]
        },
        {
            "name": "MapTextureAtlas", //Аналогично для всех текстурных атласов
            "filePath": "res/textures/eventAssets.png",
            "tileWidth": 16,
            "tileHeight": 16,
            "tiles": [

            ]
        },
        {
            "name": "GameObjectAtlas",
            "filePath": "res/textures/tankAtlas.png",
            "tileWidth": 16,
            "tileHeight": 16,

```

```

        "tiles": [
            ]
        },
    ],

    "sprites": [ //Здесь идёт разметка спрайтов, у которых есть текстура и свой шейдер
        {
            "name": "brickWall_All", //Имя спрайта
            "textureAtlas": "MiniMapTextureAtlas", //Атлас текстуры
            "shader": "SpriteShader", //Имя шейдера для спрайта
            "initialTile": "brickWall_All" //Начальная текстура. Это имя ищется в описании текстур.
        },
        {
            "name": "water",
            "textureAtlas": "MiniMapTextureAtlas",
            "shader": "SpriteShader",
            "initialTile": "water1",
            "frames": [ //Если у спрайта есть анимация, её тоже нужно обозначить. Сам спрайт имеет
начальную текстуру initialTile, но нужны и другие кадры
                {
                    "tile": "water1", //Имя первого кадра анимации. Напоминание. Все эти имена были
обозначены в разметке атласов текстур, здесь мы их просто сопоставляем
                    "duration": 500 //Длительность кадра в наносекундах
                },
                {
                    "tile": "water2", //Имя второго кадра анимации
                    "duration": 500
                },
                {
                    "tile": "water3", //Имя третьего кадра анимации
                    "duration": 500
                }
            ]
        }
    ],

    "levels": [ //Описание уровня
        {
            "description": [
                "MDDDDDNDDDDO",
                "D1D1D1D1D1D1D",
                "D4D4D4D4D4D4D",
                "D4D4D4D4D4D4D",
                "D4D4D494D4D4D",
                "DBD4D3D3D4DBD",
                "D3D3D1D1D3D3D",
                "1D11D3D3D11D1",
                "8D33D1D1D33D8",
                "D1D1D444D1D1D",
                "D4D4D4D4D4D4D",
                "D4D4D3D3D4D4D",
                "D4D4DH1GD4D4D",
                "DDDDK09s2LDDDD"
            ]
        }
    ]
}

```

Разметка ресурсов для игры уже есть в проекте, который Вы установили.

2) Создание уровня

Уровень – пространство, в котором будут располагаться игровые объекты и проходить игровой процесс, поэтому необходимо создать сначала обстановку уровня и придать им определённые свойства: координаты, физические свойства, размеры.

На уровне, который мы создадим будут располагаться объекты бетонной стены, кирпичной стены и кустов.

!Все игровые объекты необходимо создавать в папке \src\Game\GameObjects

Объекты бетонной стены и кирпичной твёрдые и могут иметь разные виды текстур, как видно на картинке “вид уровня”. Стандартный блок стены имеет размер 16 на 16 пикселей, но также есть ещё блоки стены 8 на 16 или 16 на 8. Всё это тоже нужно учесть.

Объект куста не имеет твёрдости, а также танк сможет проезжать под ним. Поэтому слой куста должен быть выше слоя отрисовки танка.

Создадим бетонную стену.

Header ConcreteWallGameObject.h

```
#pragma once

#include "GameObjectInterface.h" //Все игровые объекты должны наследовать от
GameObjectInterface
#include <memory> //для работы с указателями
#include <array> //Для работы с массивами

namespace RenderEngine {
    class Sprite; //Нам понадобится класс спрайта, поэтому обозначим его в хедере таким
    образом, а в cpp уже заинcludим. Говорят, это считается хорошим тоном.
}

class ConcreteWallGameObject : public GameObjectInterface { //Все игровые объекты должны
наследовать от GameObjectInterface
public:
    enum class WallGOType { //Для удобства создадим перечисления типов стены. Ведь для разных
типов разные текстуры.
        All,
        Top,
        Bottom,
        Left,
        Right,
        TopLeft,
        TopRight,
        BottomLeft,
        BottomRight
    };

    enum class WallGOState { //Также создадим перечисления состояний стены.
        Enabled = 0,
        Destroyed
    };

    enum class WallGOPosition { //Перечисления позиции стены
        TopLeft,
        TopRight,
        BottomLeft,
        BottomRight
    };
};
```

```

ConcreteWallGameObject(const WallGOType type, const glm::vec2& position, const glm::vec2&
size, const float rotation, const float layer); //Конструктор класса. В него понадобится
передавать тип, координаты, размеры, поворот и слой отрисовки.
virtual void Render() const override; //Метод класса GameObjectInterface. Необходим для
работы с движком и отрисовки объекта. Его нужно переопределить
virtual void UpdateFrame(const double delta) override; //Метод класса GameObjectInterface.
Необходим для работы с движком и кадровой обработки состояния объекта. Его нужно
переопределить

```

```
private:
```

```

void RenderWallGo(const WallGOPosition position) const; //Так как у нас разные типы стен,
то для каждого типа понадобится свой способ отрисовки спрайта. Разные размеры и текстуры.

```

```

std::array<WallGOState, 4> currentWallGOState; //Текущее состояние объекта. Цел или
разрушен
std::shared_ptr<RenderEngine::Sprite> sprite; //Спрайт стены
std::array<glm::vec2, 4> offsets; //Так как у нас разные типы стен, то при отрисовке
придётся смещать некоторые текстуры. За это будет отвечать массив смещений для каждого типа
};

```

СРР-реализация

```

#include "ConcreteWallGameObject.h"
#include "../Renderer/Sprite.h"
#include "../Resources/ResourceManager.h"

```

```

ConcreteWallGameObject::ConcreteWallGameObject(const WallGOType type, const glm::vec2&
position, const glm::vec2& size, const float rotation, const float layer)
: GameObjectInterface(GameObjectInterface::ObjectType::Constant_Satic_Object, position,
size, rotation, layer), currentWallGOState{ WallGOState::Destroyed, WallGOState::Destroyed,
WallGOState::Destroyed, WallGOState::Destroyed },
sprite(ResourceManager::GetSprite("betonWall")),
offsets{ glm::vec2(0, GOIsize.y / 2.f),
glm::vec2(GOIsize.x / 2.f, GOIsize.y / 2.f),
glm::vec2(0, 0),
glm::vec2(GOIsize.x / 2.f, 0) }

```

```

//В конструкторе передаём значения в GameObjectInterface. Во первых, нужно указать тип
объекта. Всего их 4. Constant_Satic_Object, Constant_Dynamic_Object,
//Temporary_Satic_Object, Temporary_Dynamic_Object.
//Подвижный, способный к взаимодействию объект, подвижный неспособный к взаимодействию
объект, неподвижный, способный к взаимодействию объект, неподвижный неспособный к
взаимодействию объект. В данном случае это Constant_Satic_Object
//Далее передаём позицию, размер, поворот и слой отрисовки, как и в базовом конструкторе.
//Инициализация sprite - ResourceManager::GetSprite("betonWall"), где betonWall - имя
текстуры и спрайта, размеченного в json файле
//Инициализация offsets - расписываем смещения для каждого типа текстуры. Переменные с
надписью GOI - переменные GameObjectInterface, которыми нужно пользоваться
{

```

```

switch (type) //Далее в конструкторе типа стены, который создаём присваиваем состояние
Enabled. Сообщаем физическому движку, что объект твёрдый (накидываем коллайдер на него) с
помощью boxColliders.emplace_back(glm::vec2(0), GOIsize);
{
case WallGOType::All:
currentWallGOState.fill(WallGOState::Enabled);
boxColliders.emplace_back(glm::vec2(0), GOIsize);
break;
case WallGOType::Top:
currentWallGOState[static_cast<size_t>(WallGOPosition::TopLeft)] =
WallGOState::Enabled;
currentWallGOState[static_cast<size_t>(WallGOPosition::TopRight)] =
WallGOState::Enabled;

```

```

        boxColliders.emplace_back(glm::vec2(0, GOIsize.y/2), GOIsize); //Для каждого
        коллайдера указываем свои координаты и размер
        break;
    case WallGOType::Bottom:
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomLeft)] =
        WallGOState::Enabled;
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomRight)] =
        WallGOState::Enabled;
        boxColliders.emplace_back(glm::vec2(0), glm::vec2(GOIsize.x, GOIsize.y / 2));
        break;
    case WallGOType::Left:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopLeft)] =
        WallGOState::Enabled;
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomLeft)] =
        WallGOState::Enabled;
        boxColliders.emplace_back(glm::vec2(0), glm::vec2(GOIsize.x / 2, GOIsize.y));
        break;
    case WallGOType::Right:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopRight)] =
        WallGOState::Enabled;
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomRight)] =
        WallGOState::Enabled;
        boxColliders.emplace_back(glm::vec2(GOIsize.x / 2, 0), GOIsize);
        break;
    case WallGOType::TopLeft:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopLeft)] =
        WallGOState::Enabled;
        boxColliders.emplace_back(glm::vec2(0, GOIsize.y / 2), glm::vec2(GOIsize.x / 2,
        GOIsize.y));
        break;
    case WallGOType::TopRight:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopRight)] =
        WallGOState::Enabled;
        boxColliders.emplace_back(glm::vec2(GOIsize.x / 2, GOIsize.y / 2), GOIsize);
        break;
    case WallGOType::BottomLeft:
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomLeft)] =
        WallGOState::Enabled;
        boxColliders.emplace_back(glm::vec2(0), glm::vec2(GOIsize.x / 2, GOIsize.y / 2));
        break;
    case WallGOType::BottomRight:
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomRight)] =
        WallGOState::Enabled;
        boxColliders.emplace_back(glm::vec2(GOIsize.x / 2, 0), glm::vec2(GOIsize.x, GOIsize.y
        / 2));
        break;
    }
}
void ConcreteWallGameObject::RenderWallGo(const WallGOPosition position) const
{
    const WallGOState state = currentWallGOState[static_cast<size_t>(position)]; //получаем
    состояние объекта
    if (state != WallGOState::Destroyed) //если он активен, отрисовываем на уровне, передвая
    позицию со смещением, размеры, поворот и слой отрисовки
    {
        sprite->Render(GOIposition + offsets[static_cast<size_t>(position)], GOIsize / 2.f,
        GOIrotation, GOIlayer); //Здесь идёт отрисовка участка стены 8*8, поэтому нужно смещение,
        которое задаётся позицией.
    }
    //Отрисовка 8 на 8 нужна, чтобы в случае отрисовки половины стены можно было не рисовать
    её половину
}
void ConcreteWallGameObject::Render() const
{

```

```

    RenderWallGo(WallGOPosition::TopLeft); //Отрисовываем каждый участок стены 8*8
    RenderWallGo(WallGOPosition::TopRight);
    RenderWallGo(WallGOPosition::BottomLeft);
    RenderWallGo(WallGOPosition::BottomRight);
}

void ConcreteWallGameObject::UpdateFrame(const double delta)
{
    //С этой стеной ничто не может произойти. Она статична на уровне, поэтому она постоянная
    каждый кадр
}

```

Аналогично создаём классы кирпичной стены и кустов/деревьев

WallGameObject.h

```

#pragma once

#include "GameObjectInterface.h"
#include <memory>
#include <array>

namespace RenderEngine {
    class Sprite;
}

class WallGameObject : public GameObjectInterface {
public:
    enum class WallGOType {
        All,
        Top,
        Bottom,
        Left,
        Right,
        TopLeft,
        TopRight,
        BottomLeft,
        BottomRight
    };

    enum class WallGOState {
        All = 0,
        TopLeft,
        TopRight,
        Top,
        BottomLeft,
        Left,
        TopRight_BottomLeft,
        Top_BottomLeft,
        BottomRight,
        TopLeft_BottomRight,
        Right,
        Top_BottomRight,
        Bottom,
        TopLeft_Bottom,
        TopRight_Bottom,
        Destroyed
    };

    enum class WallGOPosition {
        TopLeft,
        TopRight,
        BottomLeft,
        BottomRight
    };
}

```

```

    WallGameObject(const WallGOType type, const glm::vec2& position, const glm::vec2& size,
const float rotation, const float layer);
    virtual void Render() const override;
    virtual void UpdateFrame(const double delta) override;

private:
    void RenderWallGo(const WallGOPosition position) const;
    std::array<glm::vec2, 4> offsets;
    std::array<WallGOState, 4> currentWallGOState;
    std::array<std::shared_ptr<RenderEngine::Sprite>, 15> pSprites;
};

```

WallGameObject.cpp

```

#include "WallGameObject.h"
#include "../Renderer/Sprite.h"
#include "../Resources/ResourceManager.h"

WallGameObject::WallGameObject(const WallGOType type, const glm::vec2& position, const
glm::vec2& size, const float rotation, const float layer)
    : GameObjectInterface(GameObjectInterface::ObjectType::Constant_Satic_Object, position,
size, rotation, layer), currentWallGOState{ WallGOState::Destroyed, WallGOState::Destroyed,
WallGOState::Destroyed, WallGOState::Destroyed },
    offsets{ glm::vec2(0, GOIsize.y / 2.f),
             glm::vec2(GOIsize.x / 2.f, GOIsize.y / 2.f),
             glm::vec2(0, 0),
             glm::vec2(GOIsize.x / 2.f, 0) }
{
    //У кирпичной стены для каждого состояния будет свой спрайт, потому передаём их, вводя имена
разметки из gameResources.json
    pSprites[static_cast<size_t>(WallGOState::All)] =
ResourceManager::GetSprite("brickWall_All");
    pSprites[static_cast<size_t>(WallGOState::TopLeft)] =
ResourceManager::GetSprite("brickWall_TopLeft");
    pSprites[static_cast<size_t>(WallGOState::TopRight)] =
ResourceManager::GetSprite("brickWall_TopRight");
    pSprites[static_cast<size_t>(WallGOState::Top)] =
ResourceManager::GetSprite("brickWall_Top");
    pSprites[static_cast<size_t>(WallGOState::BottomLeft)] =
ResourceManager::GetSprite("brickWall_BottomLeft");
    pSprites[static_cast<size_t>(WallGOState::Left)] =
ResourceManager::GetSprite("brickWall_Left");
    pSprites[static_cast<size_t>(WallGOState::TopRight_BottomLeft)] =
ResourceManager::GetSprite("brickWall_TopRight_BottomLeft");
    pSprites[static_cast<size_t>(WallGOState::Top_BottomLeft)] =
ResourceManager::GetSprite("brickWall_Top_BottomLeft");
    pSprites[static_cast<size_t>(WallGOState::BottomRight)] =
ResourceManager::GetSprite("brickWall_BottomRight");
    pSprites[static_cast<size_t>(WallGOState::TopLeft_BottomRight)] =
ResourceManager::GetSprite("brickWall_TopLeft_BottomRight");
    pSprites[static_cast<size_t>(WallGOState::Right)] =
ResourceManager::GetSprite("brickWall_Right");
    pSprites[static_cast<size_t>(WallGOState::Top_BottomRight)] =
ResourceManager::GetSprite("brickWall_Top_BottomRight");
    pSprites[static_cast<size_t>(WallGOState::Bottom)] =
ResourceManager::GetSprite("brickWall_Bottom");
    pSprites[static_cast<size_t>(WallGOState::TopLeft_Bottom)] =
ResourceManager::GetSprite("brickWall_TopLeft_Bottom");
    pSprites[static_cast<size_t>(WallGOState::TopRight_Bottom)] =
ResourceManager::GetSprite("brickWall_TopRight_Bottom");

    switch (type)
    {

```



```

    case WallGOType::All:
        currentWallGOState.fill(WallGOState::All);
        boxColliders.emplace_back(glm::vec2(0), GOIsize);
        break;
    case WallGOType::Top:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopLeft)] = WallGOState::All;
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopRight)] = WallGOState::All;
        boxColliders.emplace_back(glm::vec2(0, GOIsize.y / 2), GOIsize);
        break;
    case WallGOType::Bottom:
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomLeft)] =
WallGOState::All;
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomRight)] =
WallGOState::All;
        boxColliders.emplace_back(glm::vec2(0), glm::vec2(GOIsize.x, GOIsize.y / 2));
        break;
    case WallGOType::Left:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopLeft)] = WallGOState::All;
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomLeft)] =
WallGOState::All;
        boxColliders.emplace_back(glm::vec2(0), glm::vec2(GOIsize.x / 2, GOIsize.y));
        break;
    case WallGOType::Right:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopRight)] = WallGOState::All;
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomRight)] =
WallGOState::All;
        boxColliders.emplace_back(glm::vec2(GOIsize.x / 2, 0), GOIsize);
        break;
    case WallGOType::TopLeft:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopLeft)] = WallGOState::All;
        boxColliders.emplace_back(glm::vec2(0, GOIsize.y / 2), glm::vec2(GOIsize.x / 2,
GOIsize.y));
        break;
    case WallGOType::TopRight:
        currentWallGOState[static_cast<size_t>(WallGOPosition::TopRight)] = WallGOState::All;
        boxColliders.emplace_back(glm::vec2(GOIsize.x / 2, GOIsize.y / 2), GOIsize);
        break;
    case WallGOType::BottomLeft:
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomLeft)] =
WallGOState::All;
        boxColliders.emplace_back(glm::vec2(0), glm::vec2(GOIsize.x / 2, GOIsize.y / 2));
        break;
    case WallGOType::BottomRight:
        currentWallGOState[static_cast<size_t>(WallGOPosition::BottomRight)] =
WallGOState::All;
        boxColliders.emplace_back(glm::vec2(GOIsize.x / 2, 0), glm::vec2(GOIsize.x, GOIsize.y
/ 2));
        break;
    }
}
void WallGameObject::RenderWallGo(const WallGOPosition position) const
{
    const WallGOState state = currentWallGOState[static_cast<size_t>(position)];
    if (state != WallGOState::Destroyed)
    {
        pSprites[static_cast<size_t>(state)]->Render(GOIposition +
offsets[static_cast<size_t>(position)], GOIsize / 2.f, GOIrotation, GOIlayer);
    }
}
void WallGameObject::Render() const
{
    RenderWallGo(WallGOPosition::TopLeft);
    RenderWallGo(WallGOPosition::TopRight);
    RenderWallGo(WallGOPosition::BottomLeft);
}

```

```

    RenderWallGo(WallGOPosition::BottomRight);
}

void WallGameObject::UpdateFrame(const double delta)
{
    }

```

Trees.h

//У деревьев только одна текстура, поэтому ни типы, ни состояния здесь не нужны

```

#pragma once

#include "GameObjectInterface.h"

#include <array>
#include <memory>

namespace RenderEngine {
    class Sprite;
}

class Trees : public GameObjectInterface {
public:

    enum class BlockPosition : uint8_t {
        TopLeft,
        TopRight,
        BottomLeft,
        BottomRight
    };

    Trees(const glm::vec2& position, const glm::vec2& size, const float rotation, const float
layer);
    virtual void Render() const override;

private:
    void RenderBlock(const BlockPosition blockLocation) const;

    std::shared_ptr<RenderEngine::Sprite> sprite;
    std::array<glm::vec2, 4> blockOffsets;
};

```

Trees.cpp

```

#include "Trees.h"

#include "../Resources/ResourceManager.h"
#include "../Renderer/Sprite.h"

Trees::Trees(const glm::vec2& position, const glm::vec2& size, const float rotation, const
float layer)
    : GameObjectInterface(GameObjectInterface::ObjectType::Constant_Static_Object, position,
size, rotation, layer)
    , sprite(ResourceManager::GetSprite("trees"))
    , blockOffsets{ glm::vec2(0, GOIsize.y / 2.f),
                    glm::vec2(GOIsize.x / 2.f, GOIsize.y / 2.f),
                    glm::vec2(0, 0),
                    glm::vec2(GOIsize.x / 2.f, 0) }
{
}

```

```

void Trees::RenderBlock(const BlockPosition blockLocation) const
{
    sprite->Render(GOIposition + blockOffsets[static_cast<size_t>(blockLocation)], GOIsize /
2.f, GOIrotation, GOIlayer);
}

void Trees::Render() const
{
    RenderBlock(BlockPosition::TopLeft);
    RenderBlock(BlockPosition::TopRight);
    RenderBlock(BlockPosition::BottomLeft);
    RenderBlock(BlockPosition::BottomRight);
}

```

Также нам потребуется граница уровня. Твёрдый объект, размещенный по краям карты

Border.h

```

#pragma once

#include "GameObjectInterface.h"

#include <array>
#include <memory>

namespace RenderEngine {
    class Sprite;
}

class Border : public GameObjectInterface {
public:

    Border(const glm::vec2& position, const glm::vec2& size, const float rotation, const float
layer);
    virtual void Render() const override;

private:
    std::shared_ptr<RenderEngine::Sprite> sprite;
};

```

Border.cpp

```

#include "Border.h"

#include "../Resources/ResourceManager.h"
#include "../Renderer/Sprite.h"

Border::Border(const glm::vec2& position, const glm::vec2& size, const float rotation, const
float layer) : GameObjectInterface(GameObjectInterface::ObjectType::Constant_Satic_Object,
position, size, rotation, layer), sprite(ResourceManager::GetSprite("borderWall"))
{
    boxColliders.emplace_back(glm::vec2(0), GOIsize);
}

void Border::Render() const
{
    sprite->Render(GOIposition, GOIsize, GOIrotation, GOIlayer);
}

```

Теперь, когда объекты созданы, можно приступить к созданию самого уровня.

Уровни необходимо создавать в папке \src\Game

Level.h

```
#pragma once

#include <vector>
#include <string>
#include <memory>
#include <glm/vec2.hpp>
#include "LevelInterface.h"

class Level : public LevelInterface //Все уровни должны наследовать от класса LevelInterface
{
public:
    Level(const std::vector<std::string>& levelMarkup); //Конструктор уровня, в который
    передаётся разметк уровня из gameResources.json

    //Кроме игровых объектов на уровне должны быть места появления игроков или ботов
    const glm::ivec2& GetPlayerRespawn_1() const
    {
        return playerRespawn1;
    }

    const glm::ivec2& GetPlayerRespawn_2() const
    {
        return playerRespawn2;
    }

    const glm::ivec2& GetEnemyRespawn_1() const
    {
        return enemyRespawn1;
    }

    const glm::ivec2& GetEnemyRespawn_2() const
    {
        return enemyRespawn2;
    }

    const glm::ivec2& GetEnemyRespawn_3() const
    {
        return enemyRespawn3;
    }

private:
    glm::ivec2 playerRespawn1; //координаты ремпавнов игрока
    glm::ivec2 playerRespawn2;
    glm::ivec2 enemyRespawn1;
    glm::ivec2 enemyRespawn2; //координаты ремпавнов врагов
    glm::ivec2 enemyRespawn3;
};
```

Level.cpp

```
#include "Level.h"
//Инкалдируем все игровые объекты, которые мы создали
#include "../Resources/ResourceManager.h"
#include "GameObjects/GameObjectInterface.h"
#include "LevelInterface.h"
#include "GameObjects/WallGameObject.h"
#include "GameObjects/ConcreteWallGameObject.h"
#include "GameObjects/Trees.h"
#include "GameObjects/Ice.h"
#include "GameObjects/Water.h"
#include "GameObjects/Eagle.h"
#include "GameObjects/Border.h"

#include <iostream>
#include <algorithm>
#include <cmath>

std::shared_ptr<GameObjectInterface> CreateGameObjectFromMarkup(const char markup, const
glm::vec2& position, const glm::vec2& size, const float rotation) //Для каждого значения
разметки уровня и его позиции нужно привязать игровой объект, который мы создали
{
    switch (markup)
    {
        case '0':
            return std::make_shared<WallGameObject>(WallGameObject::WallGOType::Right, position,
size, rotation, 0.f);
        case '1':
            return std::make_shared<WallGameObject>(WallGameObject::WallGOType::Bottom, position,
size, rotation, 0.f);
        case '2':
            return std::make_shared<WallGameObject>(WallGameObject::WallGOType::Left, position,
size, rotation, 0.f);
        case '3':
            return std::make_shared<WallGameObject>(WallGameObject::WallGOType::Top, position,
size, rotation, 0.f);
        case '4':
            return std::make_shared<WallGameObject>(WallGameObject::WallGOType::All, position,
size, rotation, 0.f);
        case '5':
            return
std::make_shared<ConcreteWallGameObject>(ConcreteWallGameObject::WallGOType::Right, position,
size, rotation, 0.f);
        case '6':
            return
std::make_shared<ConcreteWallGameObject>(ConcreteWallGameObject::WallGOType::Bottom, position,
size, rotation, 0.f);
        case '7':
            return
std::make_shared<ConcreteWallGameObject>(ConcreteWallGameObject::WallGOType::Left, position,
size, rotation, 0.f);
        case '8':
            return
std::make_shared<ConcreteWallGameObject>(ConcreteWallGameObject::WallGOType::Top, position,
size, rotation, 0.f);
        case '9':
            return
std::make_shared<ConcreteWallGameObject>(ConcreteWallGameObject::WallGOType::All, position,
size, rotation, 0.f);
        case 'A':
            return std::make_shared<Water>(position, size, rotation, 0.f);
```

```

    case 'B':
        return std::make_shared<Trees>(position, size, rotation, 1.f);
    case 'C':
        return std::make_shared<Ice>(position, size, rotation, -1.f);
    case 'E':
        return std::make_shared<Eagle>(position, size, rotation, 0.f);
    case 'G':
        return std::make_shared<WallGameObject>(WallGameObject::WallGOType::BottomLeft,
position, size, rotation, 0.f);
    case 'H':
        return std::make_shared<WallGameObject>(WallGameObject::WallGOType::BottomRight,
position, size, rotation, 0.f);
    case 'I':
        return std::make_shared<WallGameObject>(WallGameObject::WallGOType::TopLeft, position,
size, rotation, 0.f);
    case 'J':
        return std::make_shared<WallGameObject>(WallGameObject::WallGOType::TopRight,
position, size, rotation, 0.f);
    case 'D':
        return nullptr;
    default:
        std::cerr << "Unknown GameObject markup: " << markup << std::endl;
    }
    return nullptr;
}

```

```

Level::Level(const std::vector<std::string>& levelMarkup) : LevelInterface()
{
    if (levelMarkup.empty())
    {
        std::cerr << "Empty level markup!" << std::endl;
    }

    width = levelMarkup[0].length();
    height = levelMarkup.size();
    //задаем ширину и высоту уровня в пикселях
    widthPixels = static_cast<unsigned int>(width * BLOCK_SIZE);
    heightPixels = static_cast<unsigned int>(height * BLOCK_SIZE);
    //Задаём координаты спавнов
    playerRespawn1 = { BLOCK_SIZE * (width / 2 - 1), BLOCK_SIZE / 2 };
    playerRespawn2 = { BLOCK_SIZE * (width / 2 + 3), BLOCK_SIZE / 2 };
    enemyRespawn1 = { BLOCK_SIZE, BLOCK_SIZE * height - BLOCK_SIZE / 2 };
    enemyRespawn2 = { BLOCK_SIZE * (width / 2 + 1), BLOCK_SIZE * height - BLOCK_SIZE / 2 };
    enemyRespawn3 = { BLOCK_SIZE * width, BLOCK_SIZE * height - BLOCK_SIZE / 2 };

    levelObjects.reserve(width * height+4);
    unsigned int currentBottomOffset = static_cast<unsigned int>(BLOCK_SIZE * (height - 1) +
BLOCK_SIZE / 2.f);
    for (const std::string& currentRow : levelMarkup)
    {
        unsigned int currentLeftOffset = BLOCK_SIZE;
        for (const char currentElement : currentRow)
        {
            switch (currentElement)
            {
            case 'K': //Этими буквами в разметке обозначены спавны игроков и объектов
                playerRespawn1 = { currentLeftOffset, currentBottomOffset };
                levelObjects.emplace_back(nullptr);
                break;
            case 'L':
                playerRespawn2 = { currentLeftOffset, currentBottomOffset };
                levelObjects.emplace_back(nullptr);
                break;
            }
        }
    }
}

```

```

        case 'M':
            enemyRespawn1 = { currentLeftOffset, currentBottomOffset };
            levelObjects.emplace_back(nullptr);
            break;
        case 'N':
            enemyRespawn2 = { currentLeftOffset, currentBottomOffset };
            levelObjects.emplace_back(nullptr);
            break;
        case 'O':
            enemyRespawn3 = { currentLeftOffset, currentBottomOffset };
            levelObjects.emplace_back(nullptr);
            break;
        default:
            levelObjects.emplace_back(CreateGameObjectFromMarkup(currentElement,
glm::vec2(currentLeftOffset, currentBottomOffset), glm::vec2(BLOCK_SIZE, BLOCK_SIZE), 0.f));
// Если буква разметки не для спавна, то
// в объекты уровня помещаем созданные объекты с координатами
            break;
    }
    currentLeftOffset += BLOCK_SIZE; //переходим на следующую итерацию, смещая на 16
пикселей
    }
    currentBottomOffset -= BLOCK_SIZE; //Смещаемся вниз на 16 пикселей разметки,
перемещаясь на другую строку
    }
    // нижняя граница
    levelObjects.emplace_back(std::make_shared<Border>(glm::vec2(BLOCK_SIZE, 0.f),
glm::vec2(height * BLOCK_SIZE, BLOCK_SIZE / 2.f), 0.f, 0.f)); //Границы уровня

    // верхняя граница
    levelObjects.emplace_back(std::make_shared<Border>(glm::vec2(BLOCK_SIZE, height *
BLOCK_SIZE + BLOCK_SIZE / 2.f), glm::vec2(width * BLOCK_SIZE, BLOCK_SIZE / 2.f), 0.f, 0.f));

    // левая граница
    levelObjects.emplace_back(std::make_shared<Border>(glm::vec2(0.f, 0.f),
glm::vec2(BLOCK_SIZE, (height + 1) * BLOCK_SIZE), 0.f, 0.f));

    // правая граница
    levelObjects.emplace_back(std::make_shared<Border>(glm::vec2((width + 1) * BLOCK_SIZE,
0.f), glm::vec2(BLOCK_SIZE * 2.f, (height + 1) * BLOCK_SIZE), 0.f, 0.f));
}

```

3) Создание танка и реализация его стрельбы

Bullet.h – создание и реализация пули ничем не отличается от прошлых игровых объектов, кроме наличия скорости и необходимости проверки на взаимодействие с другими объектами.

```

#pragma once

#include "GameObjectInterface.h"

#include <array>
#include <memory>

namespace RenderEngine {
    class Sprite;
}

class Bullet : public GameObjectInterface {
public:
    enum class ObjectOrientation : uint8_t { //Ориентация пули

```

```

        Top,
        Bottom,
        Left,
        Right
    };

    Bullet(const double velocity, const glm::vec2& position, const glm::vec2& size, const
float layer);
    virtual void Render() const override;
    bool IsActive() const { return isActive; }
    void Fire(const glm::vec2& position, const glm::vec2& direction); //Создание пули с
ориентации на позиции танка
    virtual void OnCollision() override; //Проверка объекта на взаимодействие с другим.
Наследуемый от GameObjectInterface метод

private:
    std::shared_ptr<RenderEngine::Sprite> pSprite_top;
    std::shared_ptr<RenderEngine::Sprite> pSprite_bottom;
    std::shared_ptr<RenderEngine::Sprite> pSprite_left;
    std::shared_ptr<RenderEngine::Sprite> pSprite_right;
    ObjectOrientation objectOrientation;
    double maxVelocity; //Максимальная скорость пули
    bool isActive; // Активна ли она
    };

```

Bullet.cpp

```

#include "Bullet.h"

#include "../Resources/ResourceManager.h"
#include "../Renderer/Sprite.h"

Bullet::Bullet(const double velocity, const glm::vec2& position, const glm::vec2& size, const
float layer) : GameObjectInterface(GameObjectInterface::ObjectType::Temporary_Dynamic_Object,
position, size, 0.f, layer)

, pSprite_top(ResourceManager::GetSprite("bullet_Top"))

, pSprite_bottom(ResourceManager::GetSprite("bullet_Bottom"))

, pSprite_left(ResourceManager::GetSprite("bullet_Left"))

, pSprite_right(ResourceManager::GetSprite("bullet_Right"))

, objectOrientation(ObjectOrientation::Top)

, maxVelocity(velocity)

, isActive(false)
{
    boxColliders.emplace_back(glm::vec2(0), size);
}

void Bullet::Render() const
{
    if (isActive) {
        switch (objectOrientation)
        {
            case ObjectOrientation::Top:
                pSprite_top->Render(GOIposition, GOIsize, GOIrotation, GOIlayer);
                break;
            case ObjectOrientation::Bottom:
                pSprite_bottom->Render(GOIposition, GOIsize, GOIrotation, GOIlayer);
                break;

```



```

        case ObjectOrientation::Left:
            pSprite_left->Render(GOIposition, GOIsize, GOIrotation, GOIlayer);
            break;
        case ObjectOrientation::Right:
            pSprite_right->Render(GOIposition, GOIsize, GOIrotation, GOIlayer);
            break;
    }
}

void Bullet::Fire(const glm::vec2& position, const glm::vec2& direction)
{
    GOIposition = position;
    GOIdirection = direction;
    if (GOIdirection.x == 0.f)
    {
        objectOrientation = (GOIdirection.y < 0) ? ObjectOrientation::Bottom :
ObjectOrientation::Top;
    }
    else
    {
        objectOrientation = (GOIdirection.x < 0) ? ObjectOrientation::Left :
ObjectOrientation::Right;
    }
    isActive = true;
    SetVelocity(maxVelocity);
}
void Bullet::OnCollision()
{
    SetVelocity(0);
    isActive = false;
}

```

Теперь нужно создать танк. У него должна быть анимация, скорость, возможность взаимодействовать с другими объектами и стрелять.

TankGameObject.h

```

#pragma once
#include <memory>
#include <glm/vec2.hpp>

#include "GameObjectInterface.h"
#include "../Renderer/SpriteAnimator.h" //Танк нужно будет анимировать, поэтому инcludируем
SpriteAnimator.h
#include "../EngineSystem/Timer.h" //При появлении танка будет временная анимация для её
реализации понадобится Timer.h

namespace RenderEngine
{
    class Sprite;
}
class Bullet; //Класс пули

class TankGameObject : public GameObjectInterface
{
public:
    TankGameObject(const double maxVelocity, const glm::vec2& position, const glm::vec2&
size, const float layer);

    void Render() const override;

```

```

enum class ObjectOrientation : uint8_t
{
    Top,
    Right,
    Left,
    Bottom
};

void SetOrientation(const ObjectOrientation orientation);

void UpdateFrame(const double delta) override; //Нужен для анимации танка

double GetMaxVelocity() const;

void SetVelocity(const double velocity) override; //Наследуемы метод передачи скорости
объекту

void Fire(); //метод стрельбы

private:
    ObjectOrientation objectOrientation;

    std::shared_ptr<Bullet> pCurrentBullet;
    std::shared_ptr<RenderEngine::Sprite> pSprite_top;
    std::shared_ptr<RenderEngine::Sprite> pSprite_bottom;
    std::shared_ptr<RenderEngine::Sprite> pSprite_left;
    std::shared_ptr<RenderEngine::Sprite> pSprite_right;
    RenderEngine::SpriteAnimator spriteAnimator_top;
    RenderEngine::SpriteAnimator spriteAnimator_bottom;
    RenderEngine::SpriteAnimator spriteAnimator_left;
    RenderEngine::SpriteAnimator spriteAnimator_right;

    std::shared_ptr<RenderEngine::Sprite> pSprite_respawn;
    RenderEngine::SpriteAnimator spriteAnimator_respawn;

    Timer respawnTimer;
    double maxVelocity;
    bool isSpawning;
};

```

TankGameObject.cpp

```

#include "TankGameObject.h"
#include "../Resources/ResourceManager.h"
#include "../Renderer/Sprite.h"
//#include "../PhysicsEngine/PhysicsEngine.h"
#include "Bullet.h"

TankGameObject::TankGameObject(const double maxVelocity,
                                const glm::vec2& position,
                                const glm::vec2& size, const float layer)
    :
    GameObjectInterface(GameObjectInterface::ObjectType::Constant_Dynamic_Object, position, size,
    0.f, layer)
    ,
    objectOrientation(ObjectOrientation::Top)
    ,
    pSprite_top(ResourceManager::GetSprite("tankSprite_top"))
    ,
    pSprite_bottom(ResourceManager::GetSprite("tankSprite_bottom"))

```

```

pSprite_left(ResourceManager::GetSprite("tankSprite_left"))
pSprite_right(ResourceManager::GetSprite("tankSprite_right"))
pCurrentBullet(std::make_shared<Bullet>(0.1, GOIposition + GOIsize / 4.f, GOIsize / 2.f,
layer))
spriteAnimator_top(pSprite_top)
spriteAnimator_bottom(pSprite_bottom)
spriteAnimator_left(pSprite_left)
spriteAnimator_right(pSprite_right)
pSprite_respawn(ResourceManager::GetSprite("respawn"))
spriteAnimator_respawn(pSprite_respawn)

        , maxVelocity(maxVelocity)

        , isSpawning(true)
{
    respawnTimer.SetCallback([&]()
    {
        isSpawning = false;
    });
    respawnTimer.Start(1500);

    boxColliders.emplace_back(glm::vec2(0), GOIsize);
    PhysicsEngineManager::PhysicsEngine::AddDynamicGameObject(pCurrentBullet);
}

void TankGameObject::Render() const
{
    if (isSpawning)
    {
        pSprite_respawn->Render(GOIposition, GOIsize, GOIrotation, GOIlayer,
spriteAnimator_respawn.GetCurrentFrame());
    }
    else
    {
        switch (objectOrientation)
        {
            case TankGameObject::ObjectOrientation::Top:
                pSprite_top->Render(GOIposition, GOIsize, GOIrotation, GOIlayer,
spriteAnimator_top.GetCurrentFrame());
                break;
            case TankGameObject::ObjectOrientation::Bottom:
                pSprite_bottom->Render(GOIposition, GOIsize, GOIrotation, GOIlayer,
spriteAnimator_bottom.GetCurrentFrame());
                break;
            case TankGameObject::ObjectOrientation::Left:
                pSprite_left->Render(GOIposition, GOIsize, GOIrotation, GOIlayer,
spriteAnimator_left.GetCurrentFrame());
                break;
            case TankGameObject::ObjectOrientation::Right:
                pSprite_right->Render(GOIposition, GOIsize, GOIrotation, GOIlayer,
spriteAnimator_right.GetCurrentFrame());
                break;
        }
    }
}

```

```

        if (pCurrentBullet->IsActive())
        {
            pCurrentBullet->Render();
        }
    }

void TankGameObject::SetOrientation(const ObjectOrientation orientation)
{
    if (objectOrientation == orientation)
    {
        return;
    }
    else
    {
        objectOrientation = orientation;
        switch (objectOrientation)
        {
            case TankGameObject::ObjectOrientation::Top:
                GOIdirection.x = 0.f;
                GOIdirection.y = 1.f;
                break;
            case TankGameObject::ObjectOrientation::Left:
                GOIdirection.x = -1.f;
                GOIdirection.y = 0.f;
                break;
            case TankGameObject::ObjectOrientation::Right:
                GOIdirection.x = 1.f;
                GOIdirection.y = 0.f;
                break;
            case TankGameObject::ObjectOrientation::Bottom:
                GOIdirection.x = 0.f;
                GOIdirection.y = -1.f;
                break;
        }
    }
}

void TankGameObject::UpdateFrame(const double deltaTime)
{
    if (isSpawning)
    {
        spriteAnimator_respawn.Update(deltaTime);
        respawnTimer.Update(deltaTime);
    }
    else
    {
        if (GOIvelocity>0)
        {
            switch (objectOrientation)
            {
                case TankGameObject::ObjectOrientation::Top:
                    spriteAnimator_top.Update(deltaTime);
                    break;
                case TankGameObject::ObjectOrientation::Bottom:
                    spriteAnimator_bottom.Update(deltaTime);
                    break;
                case TankGameObject::ObjectOrientation::Left:
                    spriteAnimator_left.Update(deltaTime);
                    break;
                case TankGameObject::ObjectOrientation::Right:
                    spriteAnimator_right.Update(deltaTime);
                    break;
            }
        }
    }
}

```

```

    }
}

double TankGameObject::GetMaxVelocity() const
{
    return maxVelocity;
}

void TankGameObject::SetVelocity(const double velocity)
{
    if (!isSpawning)
    {
        GOIvelocity = velocity;
    }
}

void TankGameObject::Fire()
{
    if (!pCurrentBullet->IsActive())
    {
        pCurrentBullet->Fire(GOIposition + GOIsize / 4.f + GOIsize * GOIdirection / 4.f,
GOIdirection);
    }
}

```

4) Класс игры

Мы создали уровень и игровой объект, которым будем управлять. Теперь нужно сказать движку, что они есть и их нужно отрисовать и обрабатывать.

Для этого есть класс Game.cpp в папке \src\Game

```

#include "Game.h"
#include "../Renderer/ShaderManager.h"
#include "../Resources/ResourceManager.h"
#include "../Renderer/TextureManager.h"
#include "../Renderer/Sprite.h"
#include "../Renderer/SpriteAnimator.h"
#include "../Game/GameObjects/TankGameObject.h" //Инcludируем созданный танк
#include "../Game/GameObjects/Bullet.h" //Инcludируем созданную пулю
#include <glm/vec2.hpp>
#include <glm/mat4x4.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <iostream>
#include <GLFW/glfw3.h>
#include "Level.h" //Инcludируем созданный уровень
#include "../PhysicsEngine/PhysicsEngine.h"

Game::Game(const glm::vec2 windowSize) : gameState(GameState::Active),
windowSize(windowSize)
{
    keysForClick.fill(false); //Это разработчик трогать не должен
}

Game::~Game()
{
}

void Game::Render() //У созданных уровней и персонажей разработчик вызывает функцию
рендеринга, которые нужно отрисовать при запуске игры
{

```

```

    if (pGameObject)
    {
        pGameObject->Render();
    }
    if (pLevel)
    {
        pLevel->Render();
    }
}

void Game::Update(const double deltaTime)
{
    if (pLevel)
    {
        pLevel->Update(deltaTime); //Вызов функции покадровой обработки у уровня
    }
    if (pGameObject) {
        if (keysForClick[GLFW_KEY_W]) //Обработка нажатий на кнопки для управления танком
        {
            pGameObject->SetOrientation(TankGameObject::ObjectOrientation::Top);
            pGameObject->SetVelocity(pGameObject->GetMaxVelocity());
        }
        else if (keysForClick[GLFW_KEY_A])
        {
            pGameObject->SetOrientation(TankGameObject::ObjectOrientation::Left);
            pGameObject->SetVelocity(pGameObject->GetMaxVelocity());
        }
        else if (keysForClick[GLFW_KEY_S])
        {
            pGameObject->SetOrientation(TankGameObject::ObjectOrientation::Bottom);
            pGameObject->SetVelocity(pGameObject->GetMaxVelocity());
        }
        else if (keysForClick[GLFW_KEY_D])
        {
            pGameObject->SetOrientation(TankGameObject::ObjectOrientation::Right);
            pGameObject->SetVelocity(pGameObject->GetMaxVelocity());
        }
        else
        {
            pGameObject->SetVelocity(0);
        }
        if (pGameObject && keysForClick[GLFW_KEY_SPACE])
        {
            pGameObject->Fire();
        }
        pGameObject->UpdateFrame(deltaTime); //Вызов функции покадровой обработки у танка
    }
}

void Game::SetClick(const int key, const int action)
{
    keysForClick[key] = action; //Это разработчик трогать не должен
}

bool Game::InitGame()
{
    ResourceManager::LoadResourcesFromJSON("res/gameResources.json"); //Это разработчик
    трогать не должен

    auto pSpriteShaderProgram = ResourceManager::GetShaderManager("SpriteShader");

    if (!pSpriteShaderProgram) //Это разработчик трогать не должен
    {

```

```

        std::cerr << "Didn't found shaders :-( " << "SpriteShader" << std::endl;
    }

    pLevel = std::make_shared<Level>(ResourceManager::GetLevels()[0]); //Устанавливаем
уровень который хотим запустить. Их может быть много, если в gameResources, они были размечены
    windowSize.x = static_cast<int>(pLevel->GetLevelWidth()); //Это разработчик трогать не
должен
    windowSize.y = static_cast<int>(pLevel->GetLevelHeight()); //Это разработчик трогать
не должен
    PhysicsEngineManager::PhysicsEngine::SetCurrentLevel(pLevel); //Это разработчик
трогать не должен

    glm::mat4 projectionMatrix = glm::ortho(0.f, static_cast<float>(windowSize.x), 0.f,
static_cast<float>(windowSize.y), -100.f, 100.f); //Это разработчик трогать не должен

    pSpriteShaderProgram->UseShader(); //Это разработчик трогать не должен
    pSpriteShaderProgram->SetInt("textures", 0); //Это разработчик трогать не должен
    pSpriteShaderProgram->SetMatrix4x4("projectionMat", projectionMatrix); //Это
разработчик трогать не должен

    pGameObject = std::make_shared<TankGameObject>(0.05, pLevel->GetPlayerRespawn_1(),
glm::vec2(Level::BLOCK_SIZE, Level::BLOCK_SIZE), 0.f); //Инициализация танке по координатам
респауна
    PhysicsEngineManager::PhysicsEngine::AddDynamicGameObject(pGameObject); //Придание
танку физических свойств

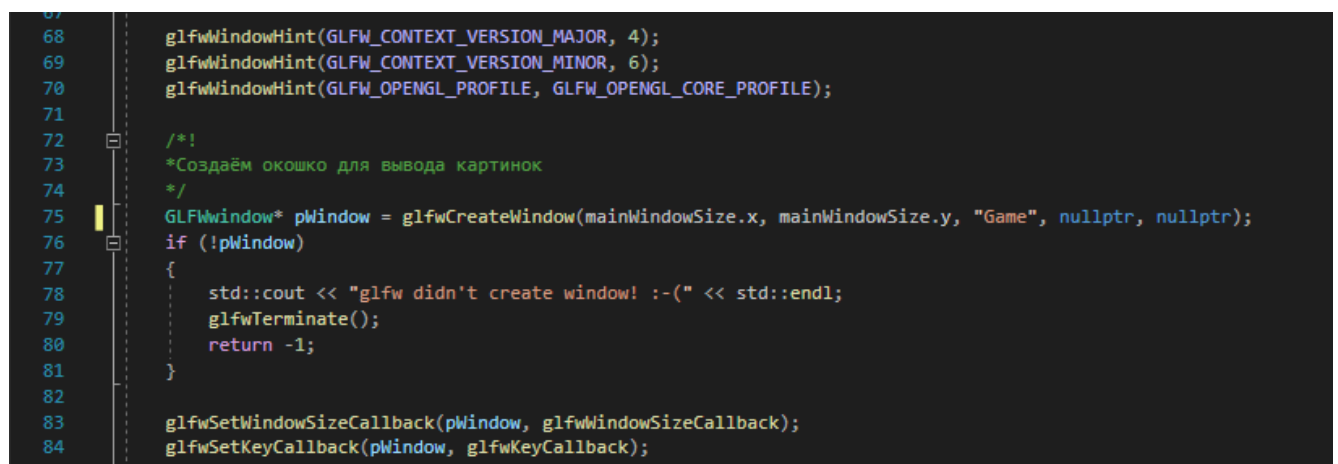
    return true;
}

size_t Game::GetCurrentLevelWidth() const
{
    return pLevel->GetLevelWidth(); //Это разработчик трогать не должен
}

size_t Game::GetCurrentLevelHeight() const
{
    return pLevel->GetLevelHeight(); //Это разработчик трогать не должен
}

```

Если нужно поменять название игры, то в main.cpp нужно поменять строку, где написано “Game” на свой текст.



```

67
68     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
69     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
70     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
71
72     /*!
73     *Создаём окошко для вывода картинок
74     */
75     GLFWwindow* pWindow = glfwCreateWindow(mainWindowSize.x, mainWindowSize.y, "Game", nullptr, nullptr);
76     if (!pWindow)
77     {
78         std::cout << "glfw didn't create window! :-( " << std::endl;
79         glfwTerminate();
80         return -1;
81     }
82
83     glfwSetWindowSizeCallback(pWindow, glfwWindowSizeCallback);
84     glfwSetKeyCallback(pWindow, glfwKeyCallback);
85

```