Game Engine Programming

Implementation Report

Maciej Legas

s4922675

Overview

VitaEngine is a 3D game engine, which has been implemented as an example Entity Component System. Its main role is to separate the game implementation functions apart from the raw underlying mechanics, such as graphics or sound, in order to simplify the process of creating a game.

VitaEngine uses a few libraries such as:

- rend, a wrapper for OpenGL rendering,
- sr1 for memory debugging,
- OpenAL for positional sound,
- STB for .ogg sound and image decoding,
- SDL2 for displaying a window.

It includes a tech demonstration, in which the player can control a flying cat model in order to test input, collisions against the solid plane, graphics and sound.

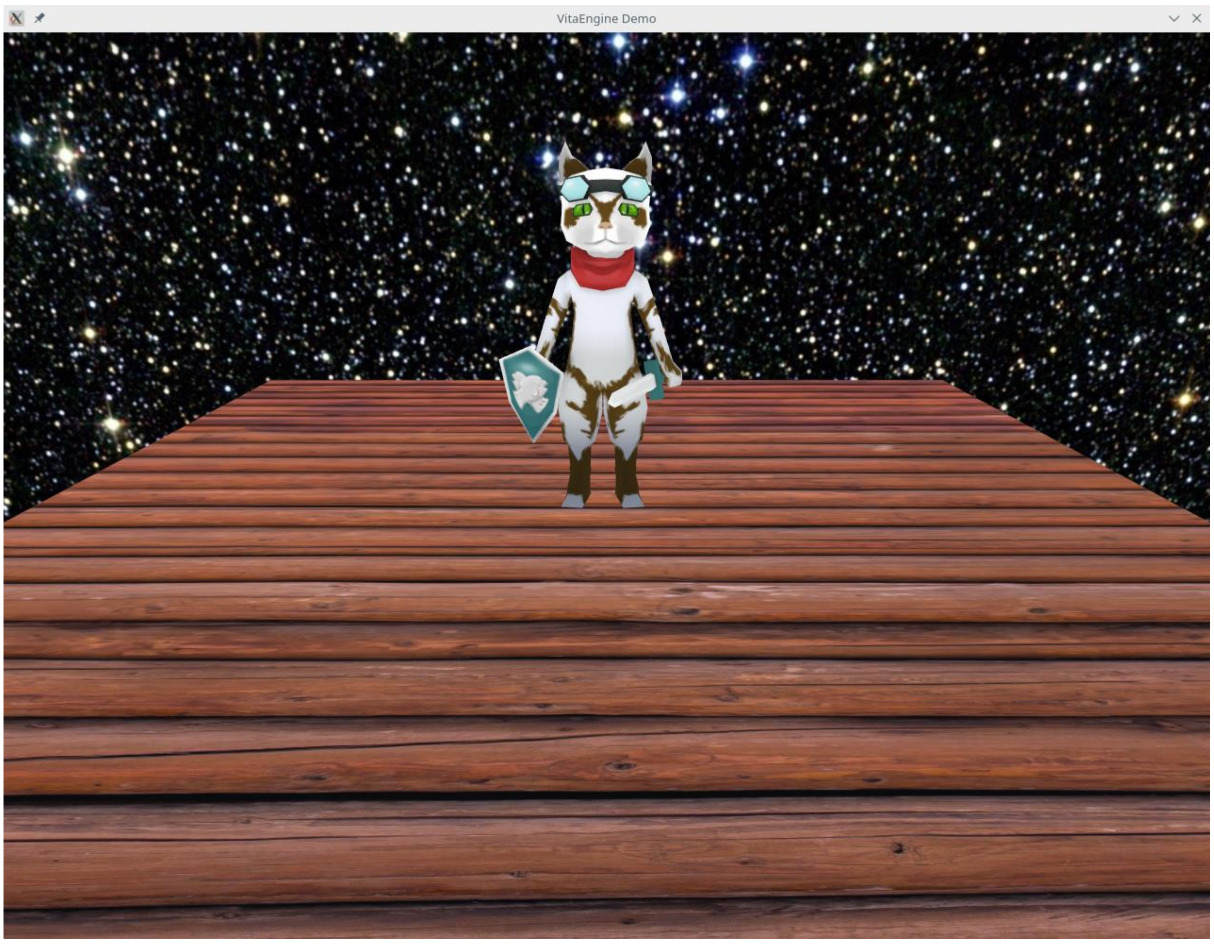


Figure 1 VitaEngine tech demonstration.

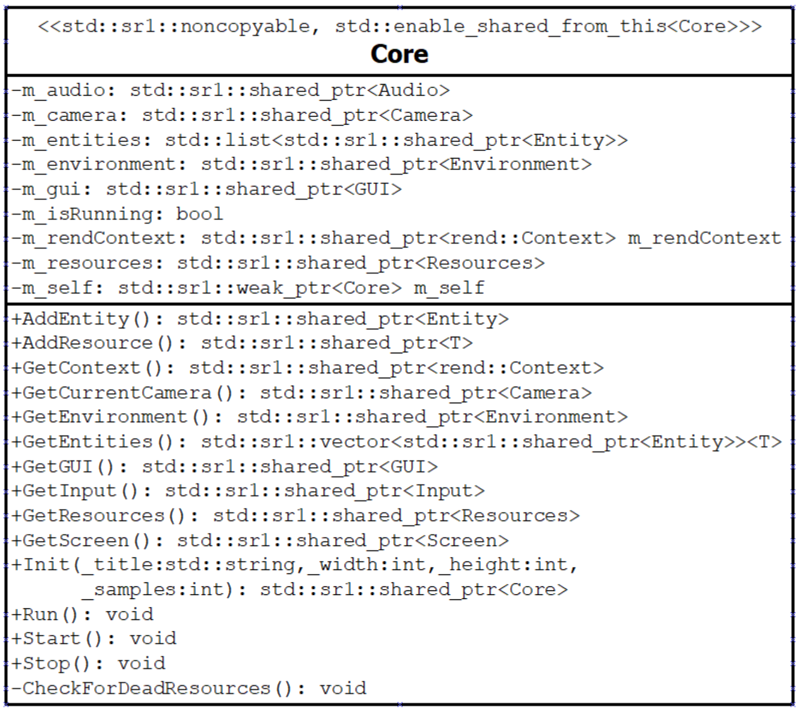
Before implementation, research into current widely used game engines has been made, such as Unreal Engine, Source Engine, or Unity. During the analysis of such, the Unity game engine, and especially its implementation of the ECS system, has been chosen for inspiration. The examples of the rend library have also been studied to understand how the wrapper works underneath, as well as a few examples from LearnOpenGL to refresh the basic knowledge of OpenGL.

In the end, whilst implementing the VitaEngine, these main features and points were held in mind:

- Taking a direct approach from the Entity Component System architecture,
- A resource loading system,
- Windows/Linux portability,
- Extensive exception checking,
- Collision detection,
- Extensive Transform class implementation,
- Large input possibility,
- GUI rendering,
- Mesh rendering.

Entity Component System

The architecture of the engine bases on a single Core class, which stores all and initializes some of the main elements of the engine, such as the resources, input or SDL window.



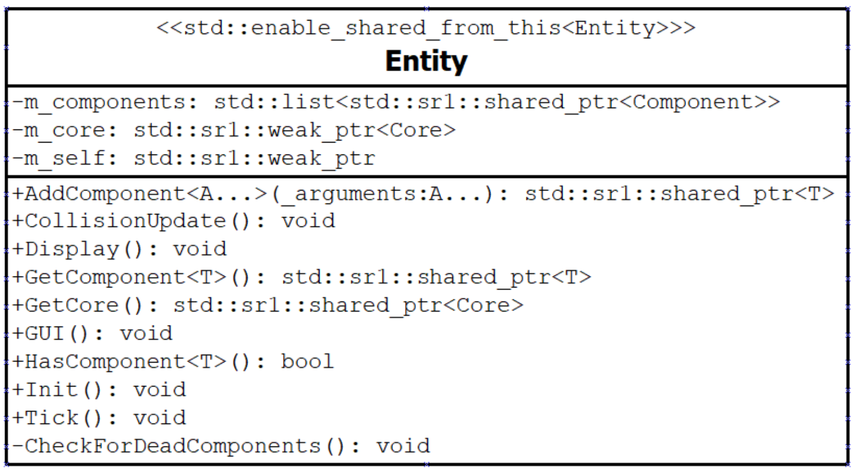
```
<<std::sr1::noncopyable, std::enable_shared_from_this<Core>>>
Core
-m_audio: std::sr1::shared_ptr<Audio>
-m_camera: std::sr1::shared_ptr<Camera>
-m_entities: std::list<std::sr1::shared_ptr<Entity>>
-m_environment: std::sr1::shared_ptr<Environment>
-m_gui: std::sr1::shared_ptr<GUI>
-m_isRunning: bool
-m_rendContext: std::sr1::shared_ptr<rend::Context> m_rendContext
-m_resources: std::sr1::shared_ptr<Resources>
-m_self: std::sr1::weak_ptr<Core> m_self
+AddEntity(): std::sr1::shared_ptr<Entity>
+AddResource(): std::sr1::shared_ptr<T>
+GetContext(): std::sr1::shared_ptr<rend::Context>
+GetCurrentCamera(): std::sr1::shared_ptr<Camera>
+GetEnvironment(): std::sr1::shared_ptr<Environment>
+GetEntities(): std::sr1::vector<std::sr1::shared_ptr<Entity>><T>
+GetGUI(): std::sr1::shared_ptr<GUI>
+GetInput(): std::sr1::shared_ptr<Input>
+GetResources(): std::sr1::shared_ptr<Resources>
+GetScreen(): std::sr1::shared_ptr<Screen>
+Init(_title:std::string,_width:int,_height:int,
      _samples:int): std::sr1::shared_ptr<Core>
+Run(): void
+Start(): void
+Stop(): void
-CheckForDeadResources(): void
```

Figure 2 The Core of the engine.

To initialize an Entity, it must be first constructed and stored in a list of Entities in the Core before returning it back to the user. This is due to three reasons:

1. To allow Entities to find other Entities containing a Component of a specified type,
2. To allow Components to access main elements of the Core,
3. To call Component functions directly in the Core's main loop.

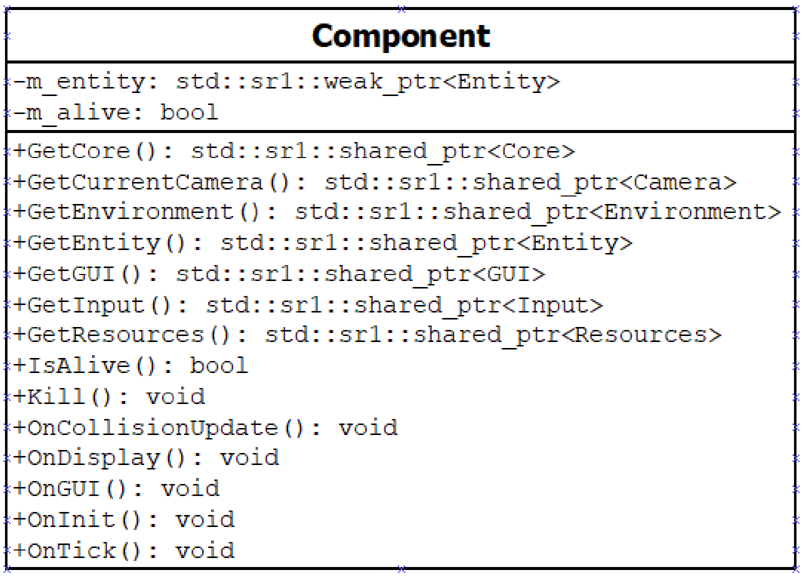
In the engine, an Entity acts as a container for Components, containing a weak pointer of the Core to allow access to the Core for Components.



```
<<std::enable_shared_from_this<Entity>>>
Entity
-m_components: std::list<std::sr1::shared_ptr<Component>>
-m_core: std::sr1::weak_ptr<Core>
-m_self: std::sr1::weak_ptr
+AddComponent<A...>(_arguments:A...): std::sr1::shared_ptr<T>
+CollisionUpdate(): void
+Display(): void
+GetComponent<T>(): std::sr1::shared_ptr<T>
+GetCore(): std::sr1::shared_ptr<Core>
+GUI(): void
+HasComponent<T>(): bool
+Init(): void
+Tick(): void
-CheckForDeadComponents(): void
```

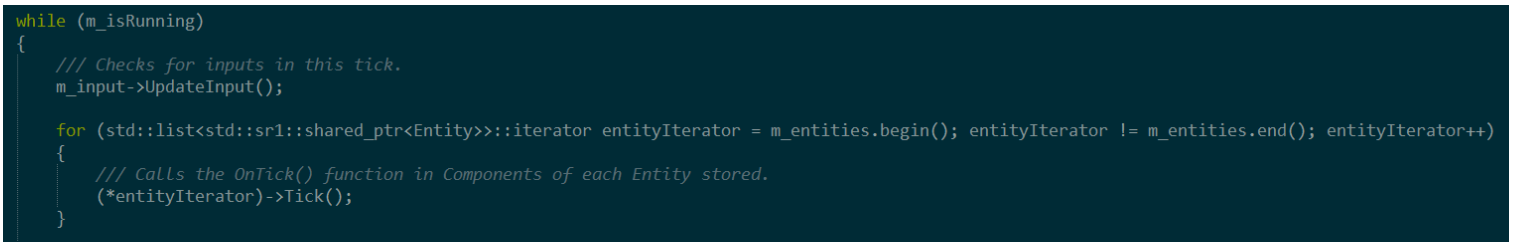
Figure 3 The Entity class.

The Component class, besides containing shortcut functions, such as GetEnvironment(), provides a few virtual functions for derived Component classes to implement, such as OnInit(), which can then be used in the Core's main loop.



```
Component
-m_entity: std::sr1::weak_ptr<Entity>
-m_alive: bool
+GetCore(): std::sr1::shared_ptr<Core>
+GetCurrentCamera(): std::sr1::shared_ptr<Camera>
+GetEnvironment(): std::sr1::shared_ptr<Environment>
+GetEntity(): std::sr1::shared_ptr<Entity>
+GetGUI(): std::sr1::shared_ptr<GUI>
+GetInput(): std::sr1::shared_ptr<Input>
+GetResources(): std::sr1::shared_ptr<Resources>
+IsAlive(): bool
+Kill(): void
+OnCollisionUpdate(): void
+OnDisplay(): void
+OnGUI(): void
+OnInit(): void
+OnTick(): void
```

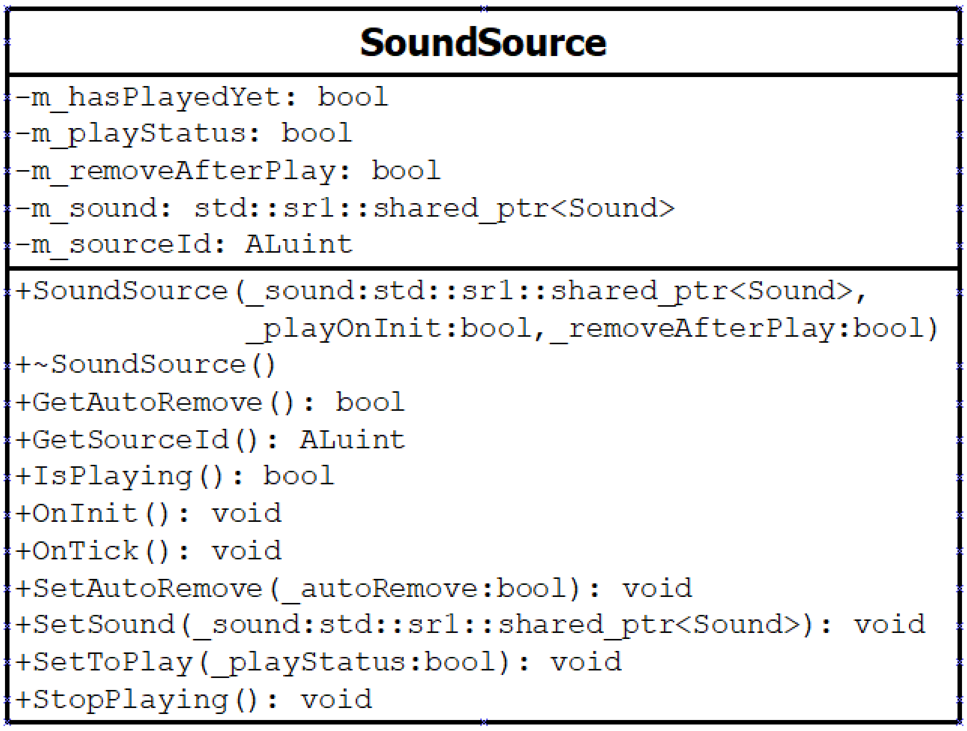
Figure 4 The Component class.



```
while (m_isRunning)
{
    /// Checks for inputs in this tick.
    m_input->UpdateInput();

    for (std::list<std::sr1::shared_ptr<Entity>>::iterator entityIterator = m_entities.begin(); entityIterator != m_entities.end(); entityIterator++)
    {
        /// Calls the OnTick() function in Components of each Entity stored.
        (*entityIterator)->Tick();
    }
```

Figure 5 How stored Entities are mainly used in the Core.

An example of a derived Component class is SoundSource, as seen on Figure 6. The constructor of this class takes use of the Entity's template function AddComponent(A… arguments) and implements its own OnInit() and OnTick() functions.

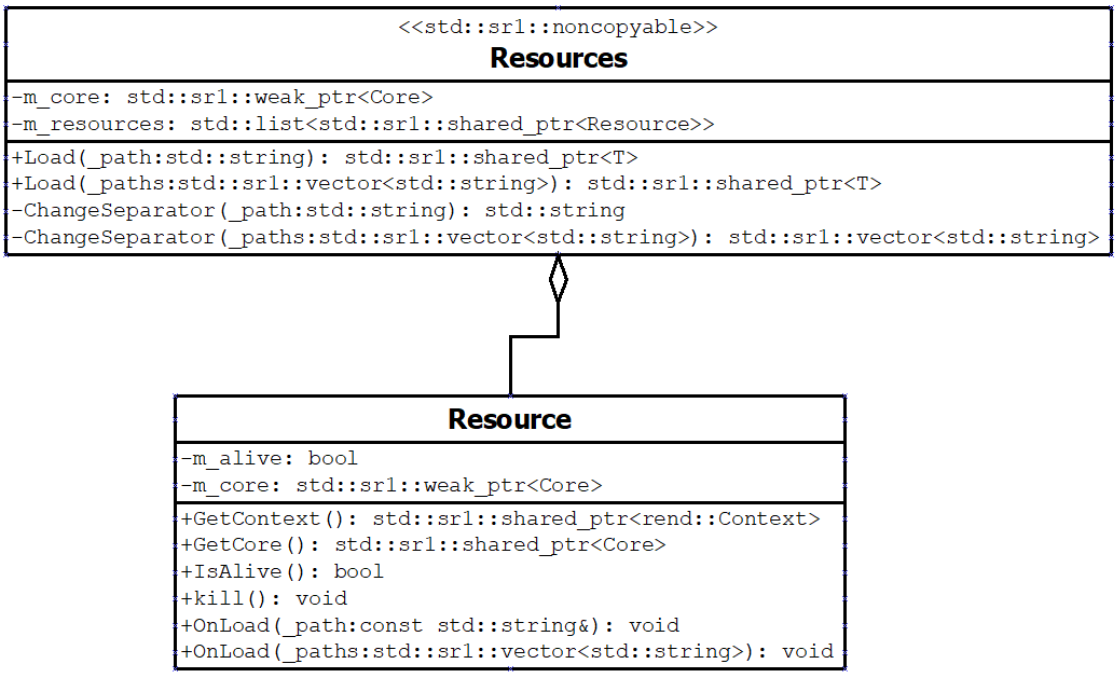


```
SoundSource
-m_hasPlayedYet: bool
-m_playStatus: bool
-m_removeAfterPlay: bool
-m_sound: std::sr1::shared_ptr<Sound>
-m_sourceId: ALuint
+SoundSource(_sound:std::sr1::shared_ptr<Sound>,
             _playOnInit:bool,_removeAfterPlay:bool)
+~SoundSource()
+GetAutoRemove(): bool
+GetSourceId(): ALuint
+IsPlaying(): bool
+OnInit(): void
+OnTick(): void
+SetAutoRemove(_autoRemove:bool): void
+SetSound(_sound:std::sr1::shared_ptr<Sound>): void
+SetToPlay(_playStatus:bool): void
+StopPlaying(): void
```

Figure 6 Example of a derived class with its own implementation of OnInit() and OnTick() functions.

Resources

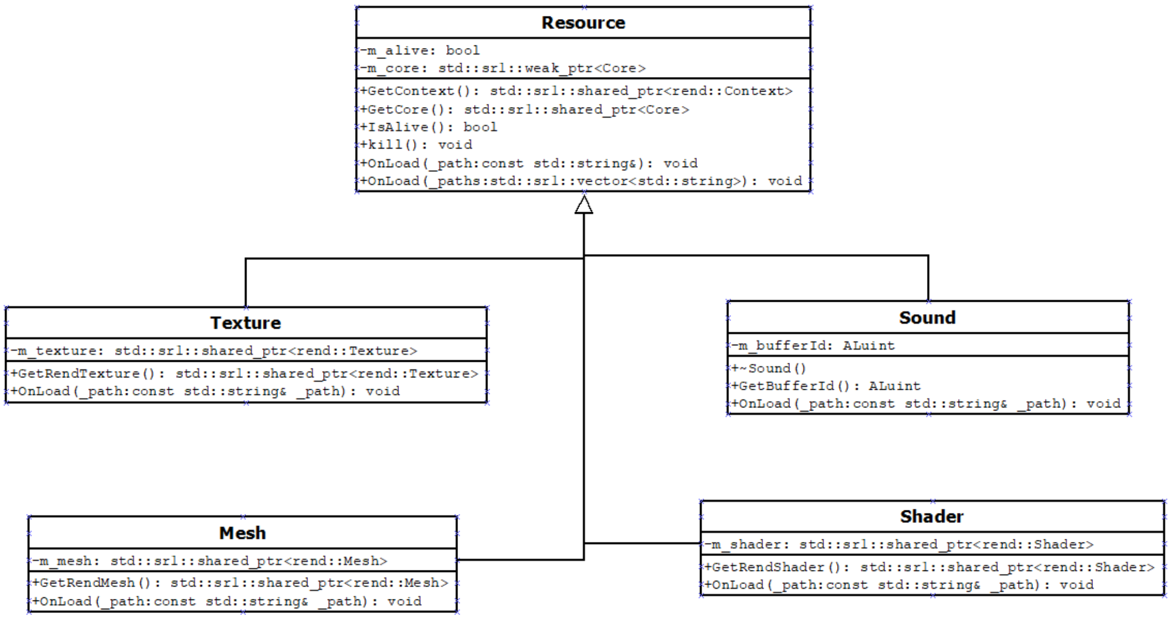
To load a resource, the Resources class is provided as the main handler of Resource classes. When the user uses the Core's AddResource() function, the loaded Resource is stored in the Resources class before returning it to the user. This allows the Core to garbage collect any no longer used Resource objects, and allows any graphical objects (such as Mesh) that require rend's Context to obtain it from Core.



<<std::sr1::noncopyable>>
Resources
-m_core: std::sr1::weak_ptr<Core>
-m_resources: std::list<std::sr1::shared_ptr<Resource>>
+Load(_path:std::string): std::sr1::shared_ptr<T>
+Load(_paths:std::sr1::vector<std::string>): std::sr1::shared_ptr<T>
-ChangeSeparator(_path:std::string): std::string
-ChangeSeparator(_paths:std::sr1::vector<std::string>): std::sr1::vector<std::string>
Resource
-m_alive: bool
-m_core: std::sr1::weak_ptr<Core>
+GetContext(): std::sr1::shared_ptr<rend::Context>
+GetCore(): std::sr1::shared_ptr<Core>
+IsAlive(): bool
+kill(): void
+OnLoad(_path:const std::string&): void
+OnLoad(_paths:std::sr1::vector<std::string>): void

Figure 7 The Resources and Resource class.

The Resources class then call the derived's Resource OnLoad() function, which parse the file depending on their type. Examples are seen on Figure 8.

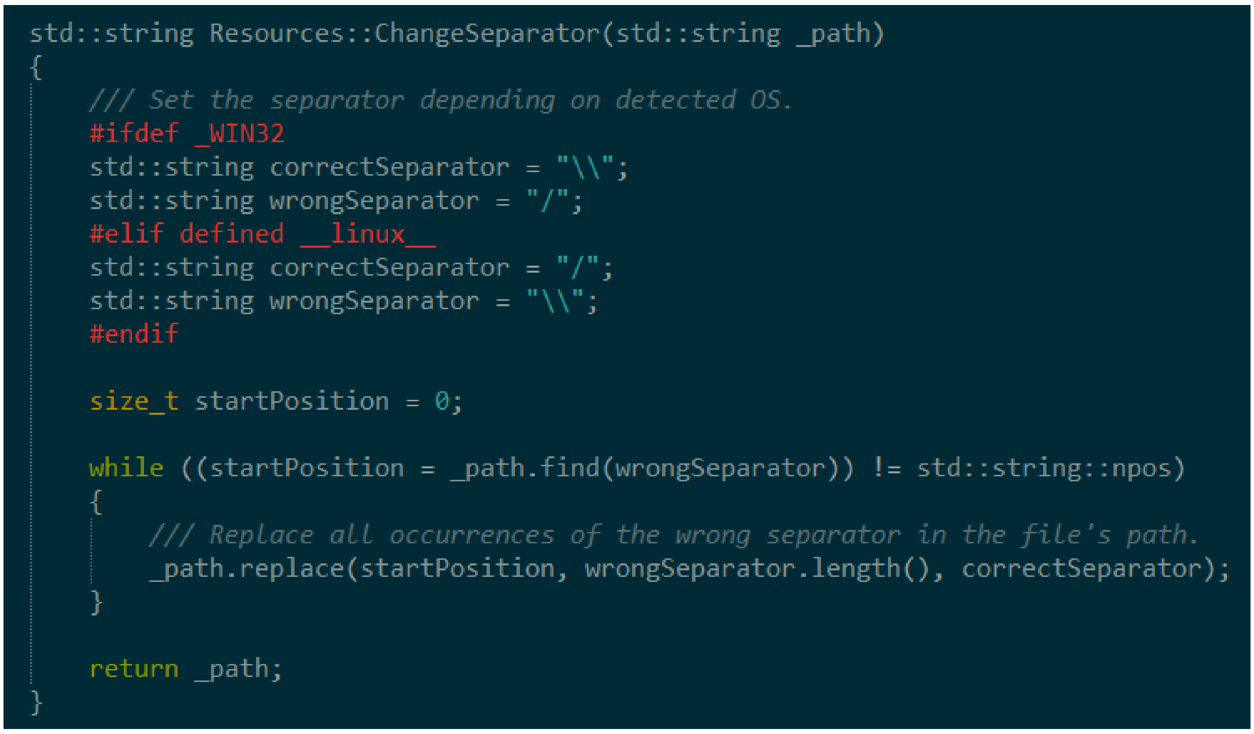


Resource
-m_alive: bool
-m_core: std::sr1::weak_ptr<Core>
+GetContext(): std::sr1::shared_ptr<rend::Context>
+GetCore(): std::sr1::shared_ptr<Core>
+IsAlive(): bool
+kill(): void
+OnLoad(_path:const std::string&): void
+OnLoad(_paths:std::sr1::vector<std::string>): void
Texture
-m_texture: std::sr1::shared_ptr<rend::Texture>
+GetRendTexture(): std::sr1::shared_ptr<rend::Texture>
+OnLoad(_path:const std::string& _path): void
Sound
-m_bufferId: ALuint
+~Sound()
+GetBufferId(): ALuint
+OnLoad(_path:const std::string& _path): void
Mesh
-m_mesh: std::sr1::shared_ptr<rend::Mesh>
+GetRendMesh(): std::sr1::shared_ptr<rend::Mesh>
+OnLoad(_path:const std::string& _path): void
Shader
-m_shader: std::sr1::shared_ptr<rend::Shader>
+GetRendShader(): std::sr1::shared_ptr<rend::Shader>
+OnLoad(_path:const std::string& _path): void

Figure 8 The derived Resource classes.

Portability

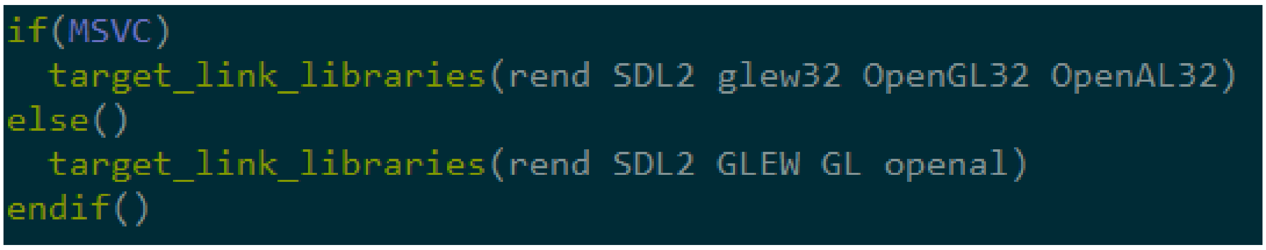
Although the engine has been coded on Linux, the portability of the project was thought of, and so a function to convert the separators in file paths has been implemented. This allows the user to specify either Windows or Linux based separators, without the worry of the compiler being unable to find the specified files in the project. This function runs automatically before loading a Resource.



```
std::string Resources::ChangeSeparator(std::string _path)
{
    /// Set the separator depending on detected OS.
    #ifdef _WIN32
    std::string correctSeparator = "\\";
    std::string wrongSeparator = "/";
    #elif defined __linux__
    std::string correctSeparator = "/";
    std::string wrongSeparator = "\\";
    #endif

    size_t startPosition = 0;

    while ((startPosition = _path.find(wrongSeparator)) != std::string::npos)
    {
        /// Replace all occurrences of the wrong separator in the file's path.
        _path.replace(startPosition, wrongSeparator.length(), correctSeparator);
    }

    return _path;
}
```

Figure 9 The separator correcting function.

The CMakeLists file has also been set to adjust to the operating system.

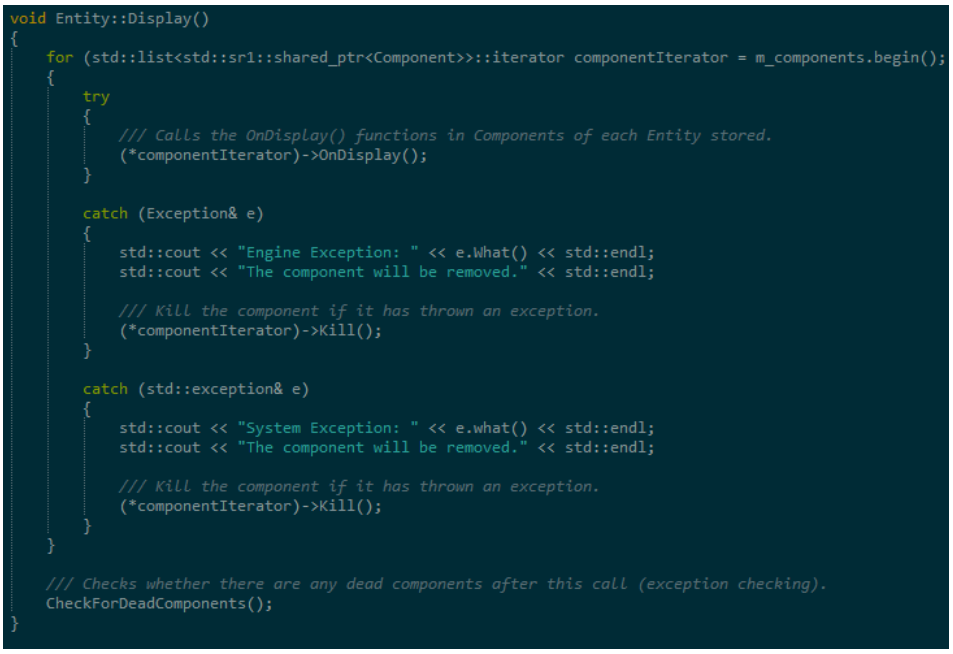


```
if(MSVC)
  target_link_libraries(rend SDL2 glew32 OpenGL32 OpenAL32)
else()
  target_link_libraries(rend SDL2 GLEW GL openal)
endif()
```

Figure 10 CMakeLists file.

Exception checking

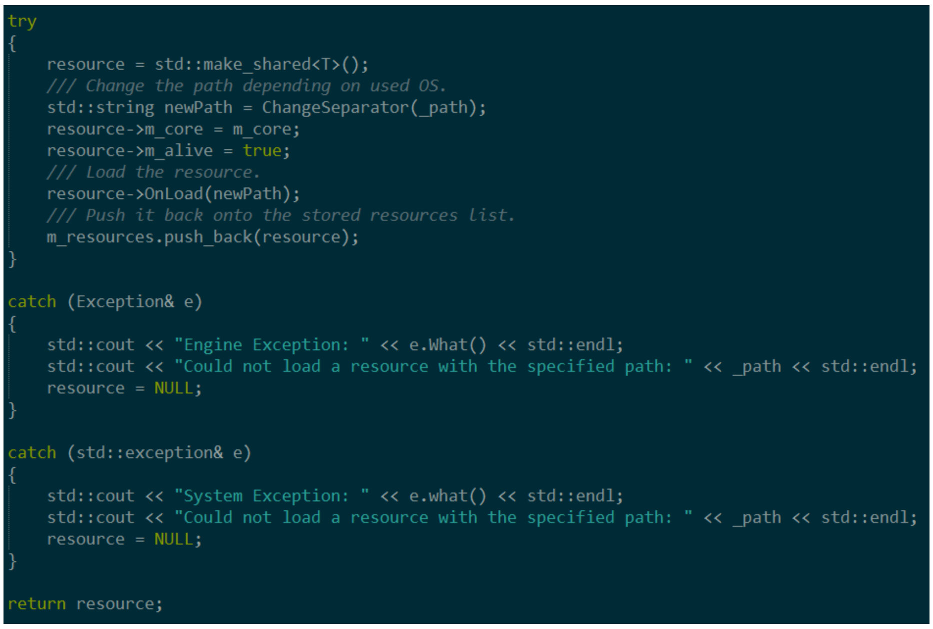
The engine involves a large amount of exception checking. As long as the exceptions thrown are not segmentation faults, which are irrecoverable, the engine can survive having all of its components throwing exceptions. In the case of a component breaking down, the engine “kills” the Component and removes it from an Entity, as shown on Figure 11. The Entities are never removed from Core, as they may have Components needed by other Components, such as a Transform.



```cpp
void Entity::Display()
{
    for (std::list<std::sr1::shared_ptr<Component>>::iterator componentIterator = m_components.begin();
    {
        try
        {
            /// Calls the OnDisplay() functions in Components of each Entity stored.
            (*componentIterator)->OnDisplay();
        }

        catch (Exception& e)
        {
            std::cout << "Engine Exception: " << e.What() << std::endl;
            std::cout << "The component will be removed." << std::endl;

            /// Kill the component if it has thrown an exception.
            (*componentIterator)->Kill();
        }

        catch (std::exception& e)
        {
            std::cout << "System Exception: " << e.what() << std::endl;
            std::cout << "The component will be removed." << std::endl;

            /// Kill the component if it has thrown an exception.
            (*componentIterator)->Kill();
        }
    }

    /// Checks whether there are any dead components after this call (exception checking).
    CheckForDeadComponents();
}
```

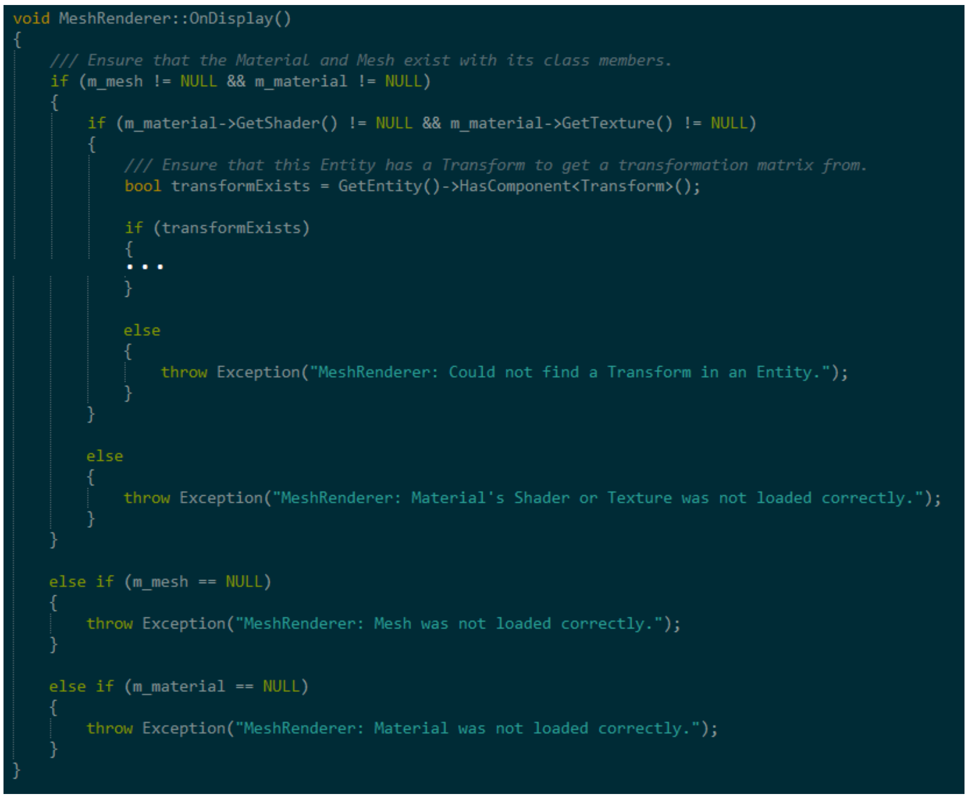
Figure 11 Component exception checking.

While loading Resources, the engine also checks for exceptions, although due to the fact that it has to return something to the user, in case of an exception it will return a NULL value. The value of a resource has to be therefore checked when using a loaded resource (Figure 12, 13).



```cpp
try
{
    resource = std::make_shared<T>();
    /// Change the path depending on used OS.
    std::string newPath = ChangeSeparator(_path);
    resource->m_core = m_core;
    resource->m_alive = true;
    /// Load the resource.
    resource->OnLoad(newPath);
    /// Push it back onto the stored resources list.
    m_resources.push_back(resource);
}

catch (Exception& e)
{
    std::cout << "Engine Exception: " << e.What() << std::endl;
    std::cout << "Could not load a resource with the specified path: " << _path << std::endl;
    resource = NULL;
}

catch (std::exception& e)
{
    std::cout << "System Exception: " << e.what() << std::endl;
    std::cout << "Could not load a resource with the specified path: " << _path << std::endl;
    resource = NULL;
}

return resource;
```

Figure 12 Loading a resource - exception checking.



```cpp
void MeshRenderer::OnDisplay()
{
    /// Ensure that the Material and Mesh exist with its class members.
    if (m_mesh != NULL && m_material != NULL)
    {
        if (m_material->GetShader() != NULL && m_material->GetTexture() != NULL)
        {
            /// Ensure that this Entity has a Transform to get a transformation matrix from.
            bool transformExists = GetEntity()->HasComponent<Transform>();

            if (transformExists)
            {
            ...
            }

            else
            {
                throw Exception("MeshRenderer: Could not find a Transform in an Entity.");
            }
        }

        else
        {
            throw Exception("MeshRenderer: Material's Shader or Texture was not loaded correctly.");
        }
    }

    else if (m_mesh == NULL)
    {
        throw Exception("MeshRenderer: Mesh was not loaded correctly.");
    }

    else if (m_material == NULL)
    {
        throw Exception("MeshRenderer: Material was not loaded correctly.");
    }
}
```

Figure 13 Exception checking for NULL.

Collision detection

The engine uses the AABB collision detection technique, using box colliders, which can have a set size and offset from the position of a Mesh. Although there has been a thought of implementing more sophisticated collision methods, such as SAT, a decision was made to refine the basic functionalities of the engine first.

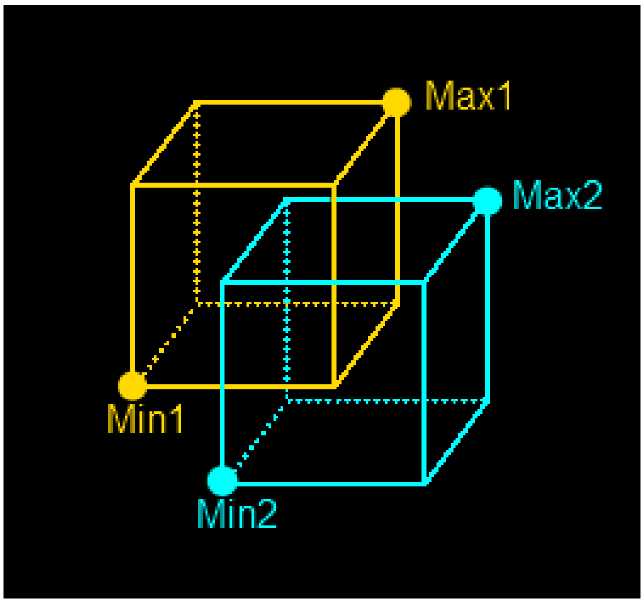
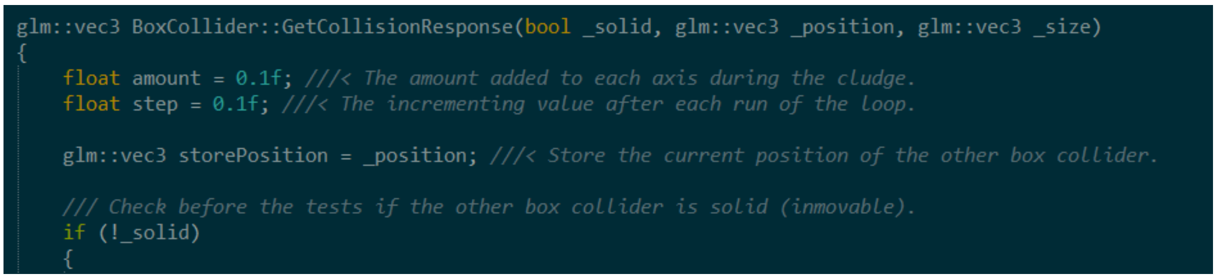


Figure 14 AABB demonstration (Casillas, 2010).

The implemented collision response has been taken from the GEP lectures ("the kludge"), although it has been modified to include solid materials, which on collision do not change their position. The example of a solid material is the plane in the demonstration.

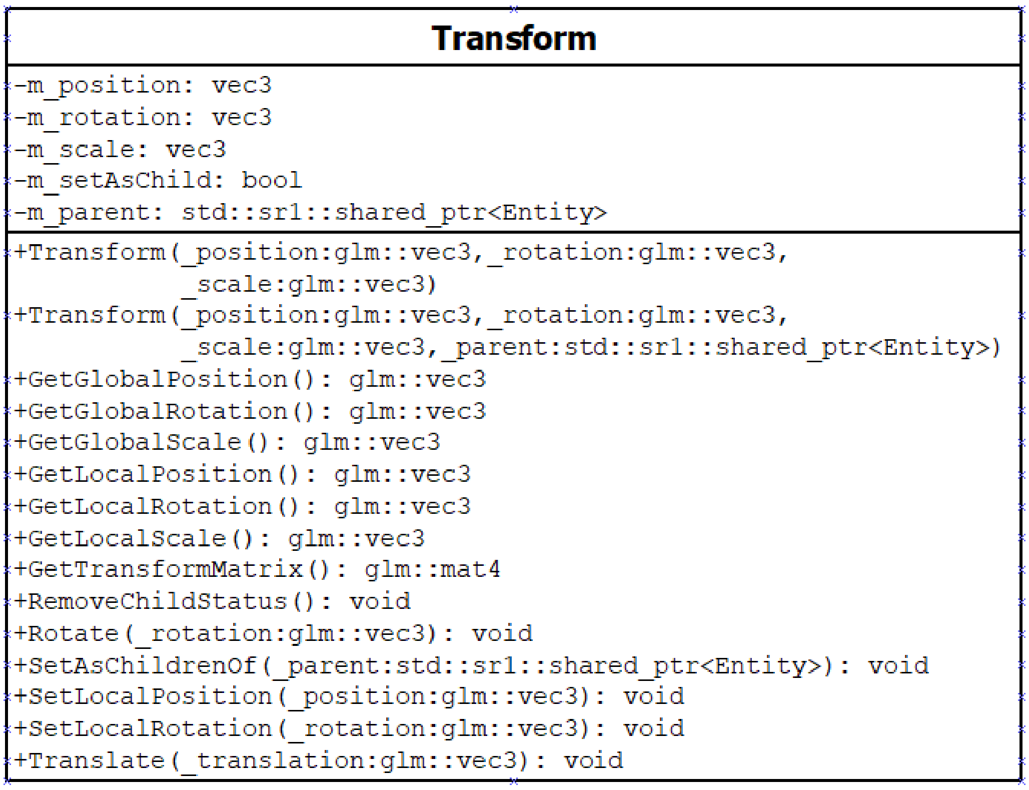


```
glm::vec3 BoxCollider::GetCollisionResponse(bool _solid, glm::vec3 _position, glm::vec3 _size)
{
    float amount = 0.1f; ///< The amount added to each axis during the cludge.
    float step = 0.1f; ///< The incrementing value after each run of the loop.

    glm::vec3 storePosition = _position; ///< Store the current position of the other box collider.

    /// Check before the tests if the other box collider is solid (inmovable).
    if (!_solid)
    {
```

Figure 15 Solid material check.

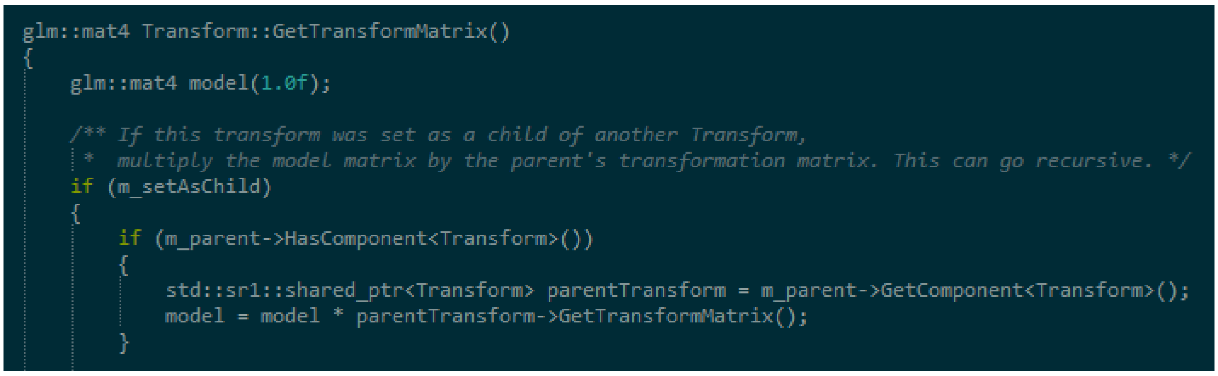
Transform

The Transform class, which stores the position, rotation and scale of an Entity, has been greatly inspired by Unity's approach, which allows to set Entities hierarchically, changing the position of all children of an Entity when the parent moves. The Transform class implemented allows for the same functionality, taking into account the transformation matrix of a parent before returning a value when using the GetGlobal…() or GetTransformMatrix() functions.



```
Transform
-m_position: vec3
-m_rotation: vec3
-m_scale: vec3
-m_setAsChild: bool
-m_parent: std::sr1::shared_ptr<Entity>
+Transform(_position:glm::vec3,_rotation:glm::vec3,
           _scale:glm::vec3)
+Transform(_position:glm::vec3,_rotation:glm::vec3,
           _scale:glm::vec3,_parent:std::sr1::shared_ptr<Entity>)
+GetGlobalPosition(): glm::vec3
+GetGlobalRotation(): glm::vec3
+GetGlobalScale(): glm::vec3
+GetLocalPosition(): glm::vec3
+GetLocalRotation(): glm::vec3
+GetLocalScale(): glm::vec3
+GetTransformMatrix(): glm::mat4
+RemoveChildStatus(): void
+Rotate(_rotation:glm::vec3): void
+SetAsChildrenOf(_parent:std::sr1::shared_ptr<Entity>): void
+SetLocalPosition(_position:glm::vec3): void
+SetLocalRotation(_rotation:glm::vec3): void
+Translate(_translation:glm::vec3): void
```

Figure 16 The Transform class.

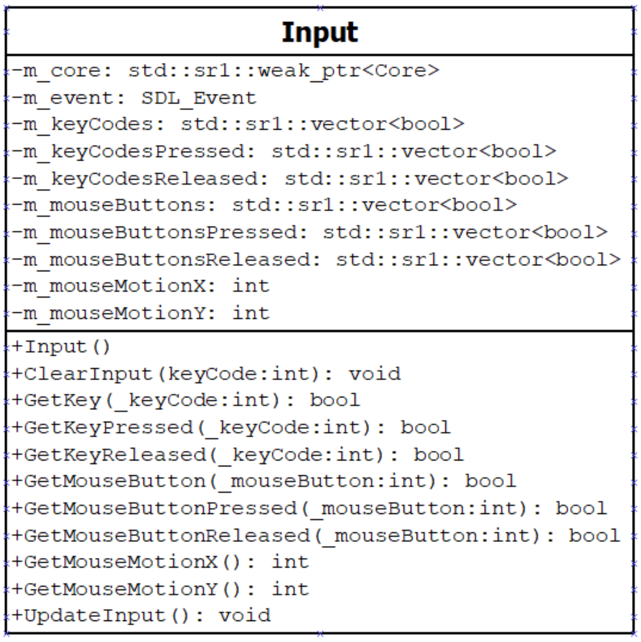


```
glm::mat4 Transform::GetTransformMatrix()
{
    glm::mat4 model(1.0f);

    /** If this transform was set as a child of another Transform,
     *  multiply the model matrix by the parent's transformation matrix. This can go recursive. */
    if (m_setAsChild)
    {
        if (m_parent->HasComponent<Transform>())
        {
            std::sr1::shared_ptr<Transform> parentTransform = m_parent->GetComponent<Transform>();
            model = model * parentTransform->GetTransformMatrix();
        }
```

Figure 17 Taking into account the position of the parent.

Input

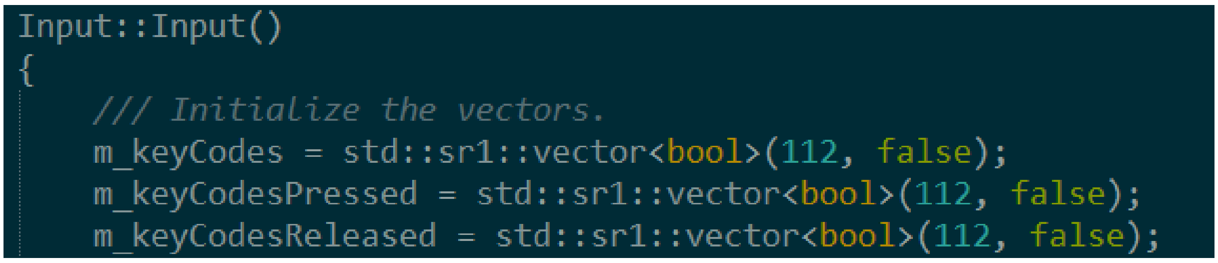
The Input class allows to process both mouse and keyboard input, storing the mouse movement, which keys/buttons were pressed and released in the current frame, as well as which keys/buttons were held.



```
Input
-m_core: std::sr1::weak_ptr<Core>
-m_event: SDL_Event
-m_keyCodes: std::sr1::vector<bool>
-m_keyCodesPressed: std::sr1::vector<bool>
-m_keyCodesReleased: std::sr1::vector<bool>
-m_mouseButtons: std::sr1::vector<bool>
-m_mouseButtonsPressed: std::sr1::vector<bool>
-m_mouseButtonsReleased: std::sr1::vector<bool>
-m_mouseMotionX: int
-m_mouseMotionY: int
+Input()
+ClearInput(keyCode:int): void
+GetKey(_keyCode:int): bool
+GetKeyPressed(_keyCode:int): bool
+GetKeyReleased(_keyCode:int): bool
+GetMouseButton(_mouseButton:int): bool
+GetMouseButtonPressed(_mouseButton:int): bool
+GetMouseButtonReleased(_mouseButton:int): bool
+GetMouseMotionX(): int
+GetMouseMotionY(): int
+UpdateInput(): void
```

Figure 18 The Input class.

A total number of 112 keycodes has been implemented to allow for most possible key combinations.



```
Input::Input()
{
    /// Initialize the vectors.
    m_keyCodes = std::sr1::vector<bool>(112, false);
    m_keyCodesPressed = std::sr1::vector<bool>(112, false);
    m_keyCodesReleased = std::sr1::vector<bool>(112, false);
```

Figure 19 Amount of keycodes.

Mesh rendering

To render a mesh, a MeshRenderer component class has been created, which takes a Material storing rend’s loaded Shader and Texture, and renders the mesh in the OnDisplay() function.

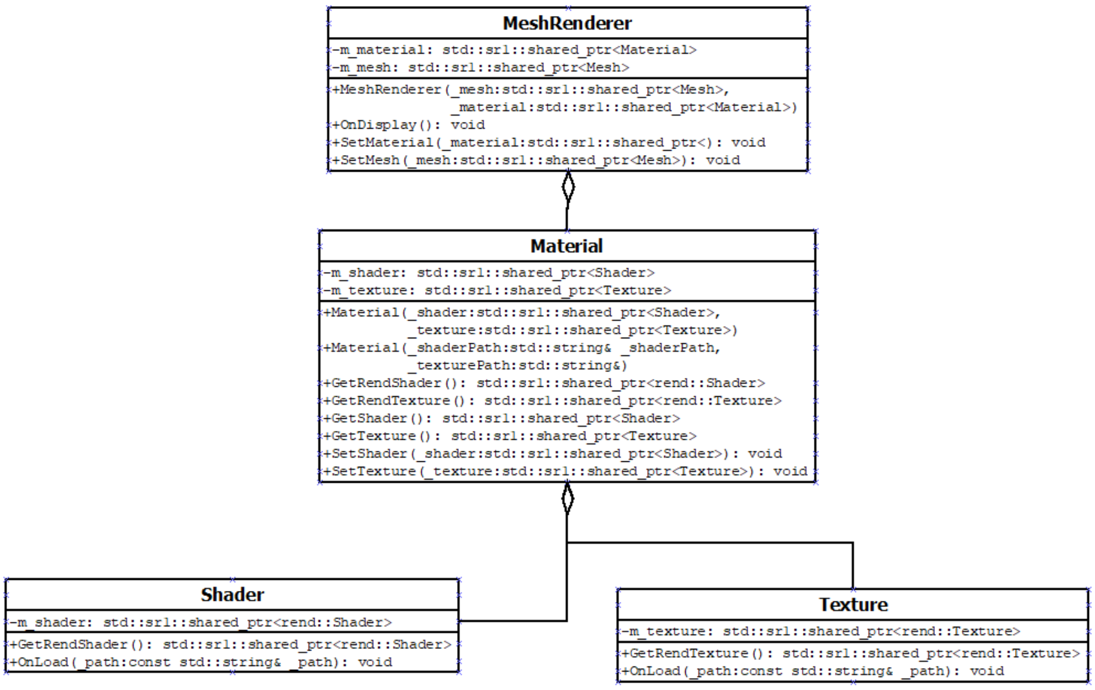
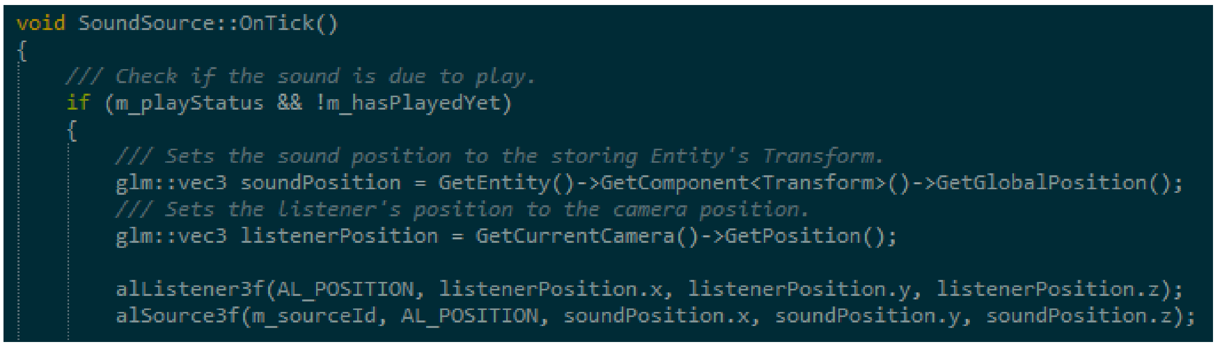

MeshRenderer
-m_material: std::srl::shared_ptr<Material>
-m_mesh: std::srl::shared_ptr<Mesh>
+MeshRenderer(_mesh:std::srl::shared_ptr<Mesh>, _material:std::srl::shared_ptr<Material>)
+OnDisplay(): void
+SetMaterial(_material:std::srl::shared_ptr<): void
+SetMesh(_mesh:std::srl::shared_ptr<Mesh>): void
Material
-m_shader: std::srl::shared_ptr<Shader>
-m_texture: std::srl::shared_ptr<Texture>
+Material(_shader:std::srl::shared_ptr<Shader>, _texture:std::srl::shared_ptr<Texture>)
+Material(_shaderPath:std::string& _shaderPath, _texturePath:std::string&)
+GetRendShader(): std::srl::shared_ptr<rend::Shader>
+GetRendTexture(): std::srl::shared_ptr<rend::Texture>
+GetShader(): std::srl::shared_ptr<Shader>
+GetTexture(): std::srl::shared_ptr<Texture>
+SetShader(_shader:std::srl::shared_ptr<Shader>): void
+SetTexture(_texture:std::srl::shared_ptr<Texture>): void
Shader
-m_shader: std::srl::shared_ptr<rend::Shader>
+GetRendShader(): std::srl::shared_ptr<rend::Shader>
+OnLoad(_path:const std::string& _path): void
Texture
-m_texture: std::srl::shared_ptr<rend::Texture>
+GetRendTexture(): std::srl::shared_ptr<rend::Texture>
+OnLoad(_path:const std::string& _path): void

Figure 20 MeshRenderer class.

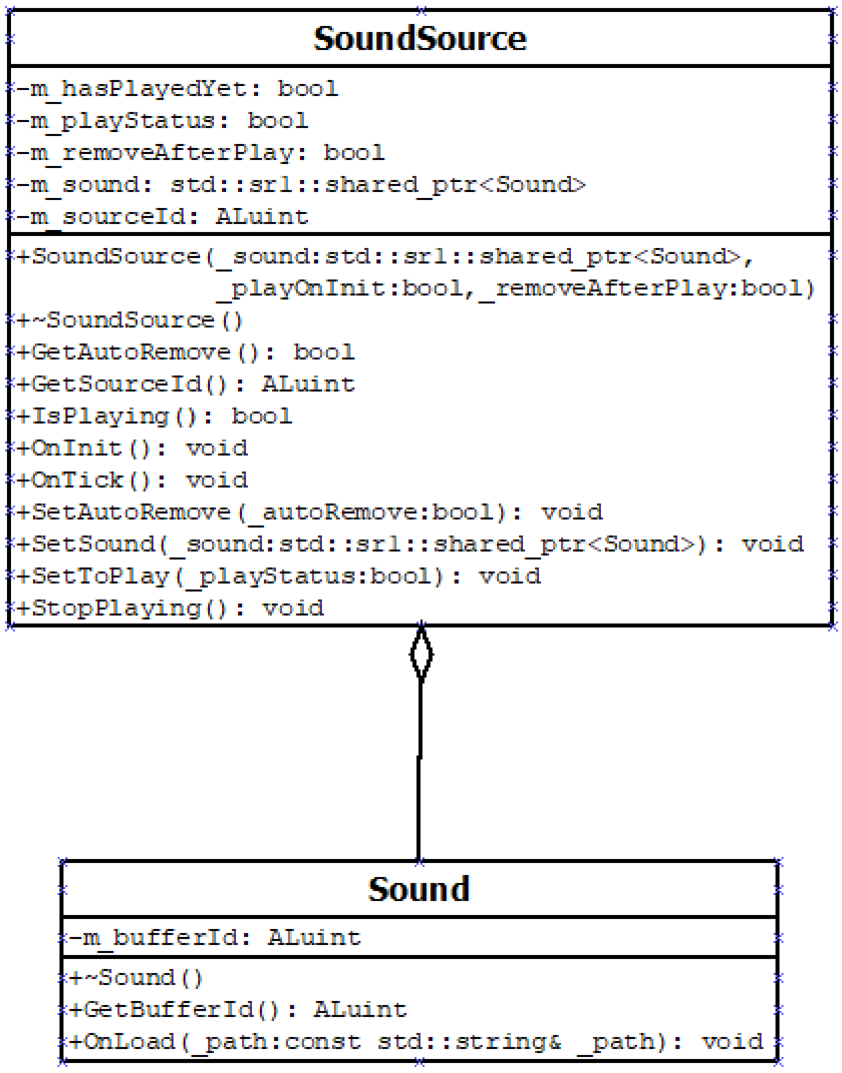
Positional audio

Using the OpenAL library, the goal of positional audio has been achieved, as it allows to set the position of a sound source to the storing Entity’s position, and the listener’s position to the current Camera rendering.



```
void SoundSource::OnTick()
{
    /// Check if the sound is due to play.
    if (m_playStatus && !m_hasPlayedYet)
    {
        /// Sets the sound position to the storing Entity's Transform.
        glm::vec3 soundPosition = GetEntity()->GetComponent<Transform>()->GetGlobalPosition();
        /// Sets the listener's position to the camera position.
        glm::vec3 listenerPosition = GetCurrentCamera()->GetPosition();

        alListener3f(AL_POSITION, listenerPosition.x, listenerPosition.y, listenerPosition.z);
        alSource3f(m_sourceId, AL_POSITION, soundPosition.x, soundPosition.y, soundPosition.z);
```

Figure 21 Demonstration of use of positional audio.

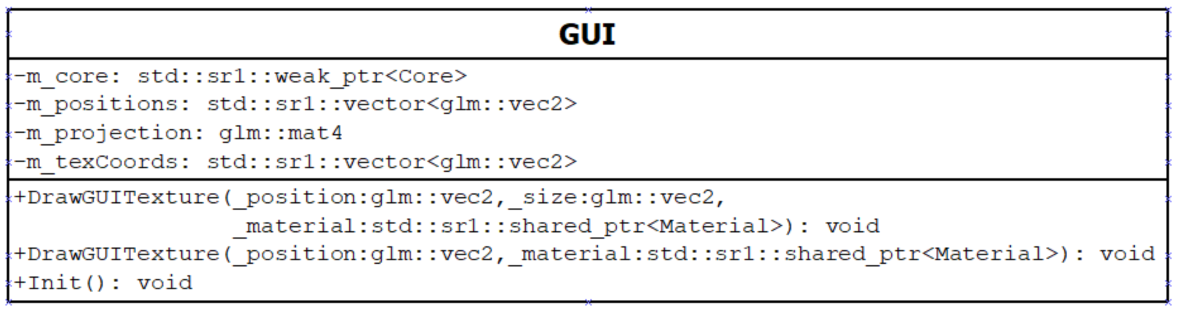


```
SoundSource
-m_hasPlayedYet: bool
-m_playStatus: bool
-m_removeAfterPlay: bool
-m_sound: std::sr1::shared_ptr<Sound>
-m_sourceId: ALuint
+SoundSource(_sound:std::sr1::shared_ptr<Sound>,
             _playOnInit:bool,_removeAfterPlay:bool)
+~SoundSource()
+GetAutoRemove(): bool
+GetSourceId(): ALuint
+IsPlaying(): bool
+OnInit(): void
+OnTick(): void
+SetAutoRemove(_autoRemove:bool): void
+SetSound(_sound:std::sr1::shared_ptr<Sound>): void
+SetToPlay(_playStatus:bool): void
+StopPlaying(): void

Sound
-m_bufferId: ALuint
+~Sound()
+GetBufferId(): ALuint
+OnLoad(_path:const std::string& _path): void
```

Figure 21 The architecture of Sound and SoundSource class.

GUI

To render 2D textures on screen, a GUI class has been implemented, which takes a Material with a loaded GUI shader and 2D texture, and renders the texture using an orthographic projection.



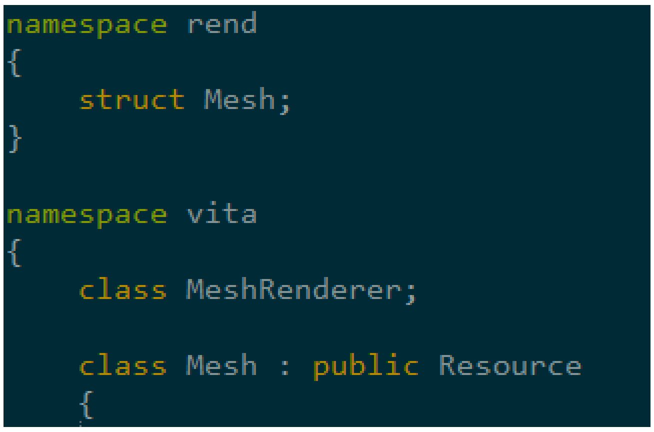
```
GUI
-m_core: std::sr1::weak_ptr<Core>
-m_positions: std::sr1::vector<glm::vec2>
-m_projection: glm::mat4
-m_texCoords: std::sr1::vector<glm::vec2>
+DrawGUITexture(_position:glm::vec2,_size:glm::vec2,
                _material:std::sr1::shared_ptr<Material>): void
+DrawGUITexture(_position:glm::vec2,_material:std::sr1::shared_ptr<Material>): void
+Init(): void
```

Figure 22 GUI class.

Evaluation

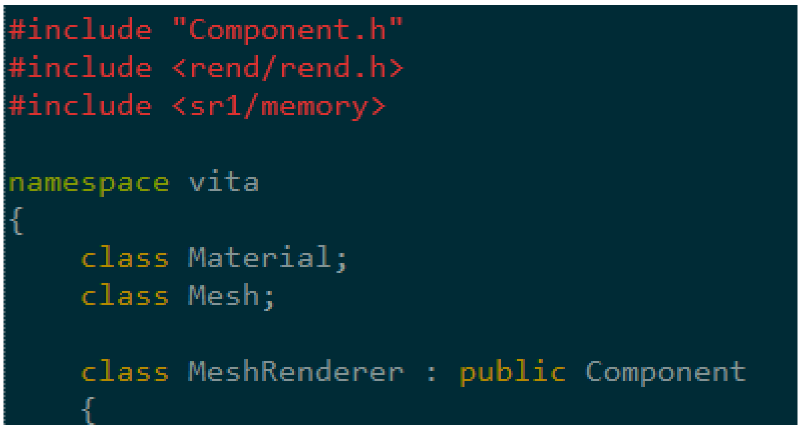
Implementation problems with rend

Initially, the approach taken was to forward declare any used rend's members in headers to ensure no OpenGL conflicts. This, however, resulted in segmentation faults when loading rend's members, such as Mesh or Texture. A decision was then made to declare rend in every header that used the library. Although this resolved the segmentation faults, it also resulted nothing rendering on the screen. As the rend library was not throwing any exceptions in the process, determining the culprit was heavily undermined. In the end, each header file that used rend was tested whether it crashes when using a forward declaration, and finally some header files were left with forward declarations and some with includes.



```
namespace rend
{
    struct Mesh;
}

namespace vita
{
    class MeshRenderer;

    class Mesh : public Resource
    {
```

Figure 23 Forward declaration of rend.

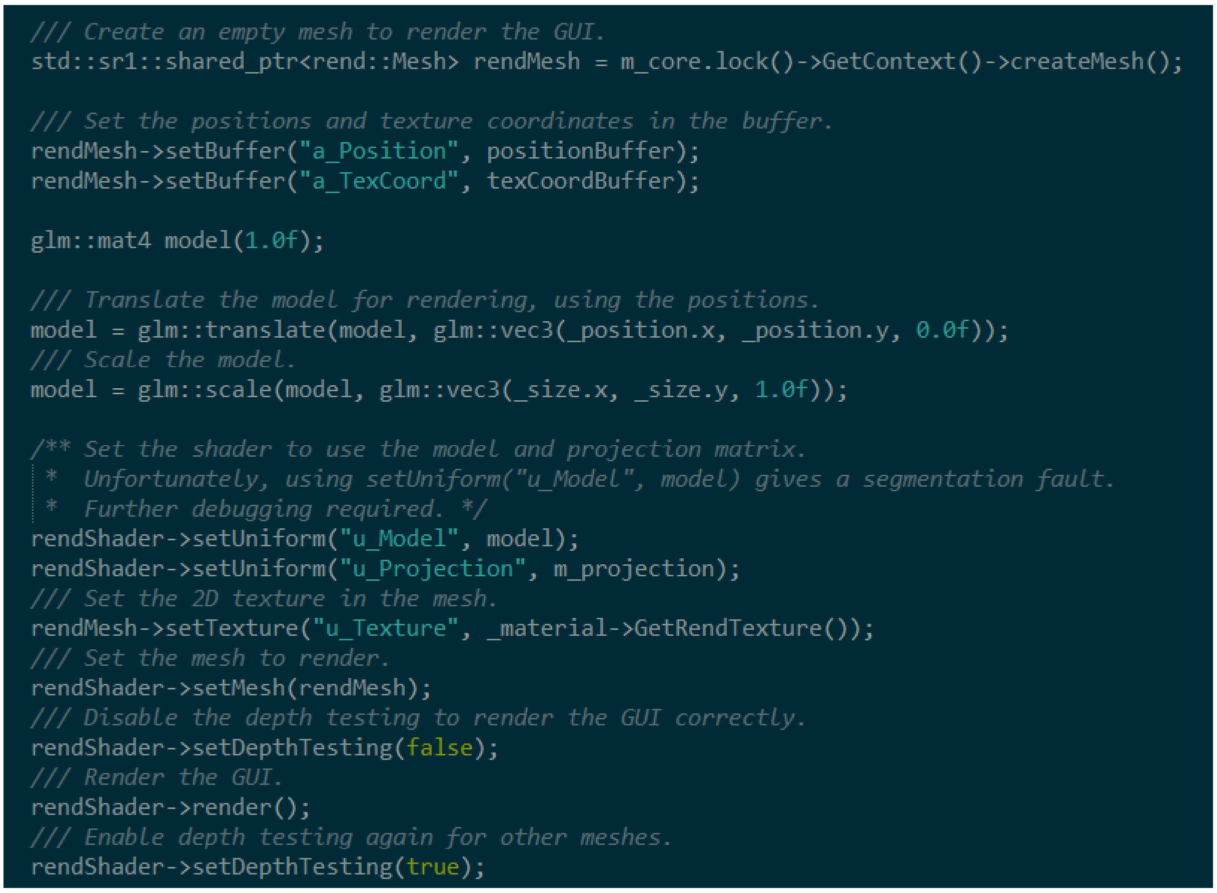


```
#include "Component.h"
#include <rend/rend.h>
#include <sr1/memory>

namespace vita
{
    class Material;
    class Mesh;

    class MeshRenderer : public Component
    {
```

Figure 24 Including rend.

GUI issues

Unfortunately, the GUI had to be left unfinished. During the display of a GUI 2D texture, setting the model transformation matrix results in a segmentation fault exception. A possible solution would be to separate the GUI from the engine and test it solely with rend to debug what is the issue, as it is a possible OpenGL buffer conflict.



```
/// Create an empty mesh to render the GUI.
std::sr1::shared_ptr<rend::Mesh> rendMesh = m_core.lock()->GetContext()->createMesh();

/// Set the positions and texture coordinates in the buffer.
rendMesh->setBuffer("a_Position", positionBuffer);
rendMesh->setBuffer("a_TexCoord", texCoordBuffer);

glm::mat4 model(1.0f);

/// Translate the model for rendering, using the positions.
model = glm::translate(model, glm::vec3(_position.x, _position.y, 0.0f));
/// Scale the model.
model = glm::scale(model, glm::vec3(_size.x, _size.y, 1.0f));

/** Set the shader to use the model and projection matrix.
 *  Unfortunately, using setUniform("u_Model", model) gives a segmentation fault.
 *  Further debugging required. */
rendShader->setUniform("u_Model", model);
rendShader->setUniform("u_Projection", m_projection);
/// Set the 2D texture in the mesh.
rendMesh->setTexture("u_Texture", _material->GetRendTexture());
/// Set the mesh to render.
rendShader->setMesh(rendMesh);
/// Disable the depth testing to render the GUI correctly.
rendShader->setDepthTesting(false);
/// Render the GUI.
rendShader->render();
/// Enable depth testing again for other meshes.
rendShader->setDepthTesting(true);
```

Figure 25 The GUI implementation.

Conclusion

Overall, the engine makes good use of the ECS architecture, and separates the engine from the game mechanics excellently. Although the engine may lack in advanced features, it has greatly expanded the base functionalities and has an extensive use of exception checking.

The possible road to improvement would be:

- Implementing a better collision technique, such as SAT, to reject the need for multiple types of colliders.
- Allowing the user to more easily change the collision response. Currently, the easiest way to do so is to create a derived class from the BoxCollider and overriding its collision functions.
- Integrating light and shadows into the engine, as currently both shadows and light positions need to be implemented directly in shaders, which is not an user friendly approach.
- Implementing the support of text, such as with the use of STB TrueType, to reduce the need and amount of 2D textures in case of implementing a heavily GUI-based game.

References

Figure 14: Casillas M. (2010). *AABB to AABB.* [online] Available at: http://www.miguelcasillas.com/?p=30 [Accessed 24 Jan 2020].

Palindrom. *Quaternion to axis angles.* (2015). [online] Available at: https://stackoverflow.com/questions/29229611/quaternion-to-axis-angles [Accessed 23 Jan 2020].

valmo. *glm – Decompose mat4 into translation and rotation?* (2015). [online] Available at: https://stackoverflow.com/questions/17918033/glm-decompose-mat4-into-translation-and-rotation [Accessed 23 Jan 2020].

Plane texture: themuralstore.com (c.a. 2017) Log Cabin (Rustic Oak) CANVAS Peel and Stick Wall Mural. [online] Available at: https://themuralstore.com/log-cabin-rustic-oak-canvaspeel-and-stick-wall-mural-dt3501.html#product-details-tab-specification [Accessed 23 Jan 2020].

Skybox texture: wallsfield.com (2015). Stars Texture Free HD Wallpapers. [online] Available at: https://wallsfield.com/stars-texture-hd-wallpapers/ [Accessed 23 Jan 2020].

Music: Sirius Beat (2013). One. [online] Available at: https://www.youtube.com/watch?v=foKOqaimVv4 [Accessed 23 Jan 2020]