3D Graphics Programming

Report

Maciej Legas

S4922675

Overview

Evade the Boxes is a simple 3D game, where the player controls a first person character. The player begins on a plane, from which he cannot fall off, set in a skybox showing a space setting, allowing him to firstly freely experiment with the environment and controls. After the player inputs a key to start the game, the game starts to endlessly spawn, with a set time interval, textured crates from above. With the fall starting, the main objective of the game begins as well, which is to avoid the endlessly spawning boxes while controlling a first person camera by user input.

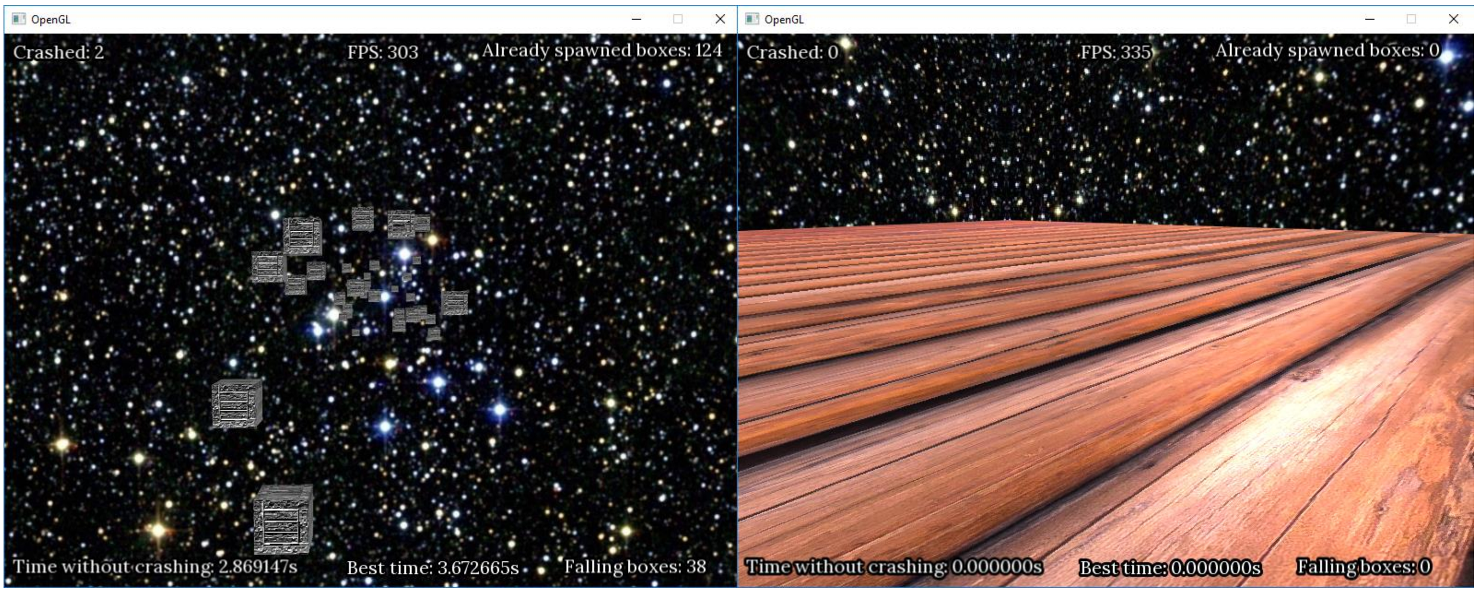

Figure 1 Two screenshots of the game.

The game checks for each crate if it collided with the player, and if any of them did, it plays an audio cue to inform the player of the collision, as well as update the amount of collisions on the HUD.

I decided not to limit the player with a time and/or life limit, and instead made the game run as long as the player wishes to continue, possibly going for own personal goals such as, for example, setting a personal time record.

Of course, as the setting is based in space, both the plane and the crates will be well-lit to distinguish them from the blackness of the skybox.

As mentioned with the time record, the HUD should display at least the current time without hitting the crates. I decided to expand on that and add a few extra information to the HUD, including an FPS counter, amount of boxes that have been already spawned, current amount of boxes in the game, and the best time achieved during the current game session.

OpenGL is in use to render the shapes and the camera view (projection and view matrices). As for the window, audio, input and HUD functionalities such as text drawing, I decided to use SFML.

Below is a small diagram showing the most basic operations the game should do.

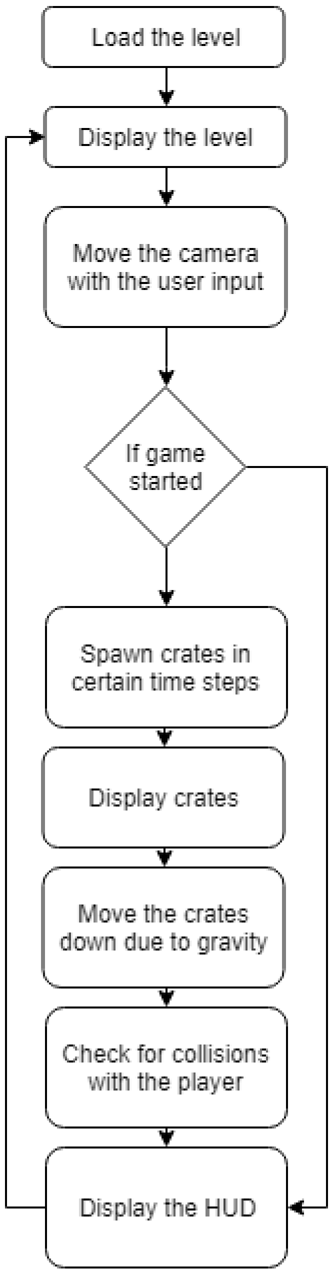
Load the level
Display the level
Move the camera with the user input
If game started
Spawn crates in certain time steps
Display crates
Move the crates down due to gravity
Check for collisions with the player
Display the HUD

Figure 2 Basic overview of the game.

Research

To detect the collisions with the player, a collision detection algorithm must be implemented. I decided to use Axis-Aligned Bounding Box collision detection, as it is more precise than sphere-sphere collision detection, and faster than the Oriented Bounding Box algorithm, since we do not need to adjust the bounding boxes with rotations.

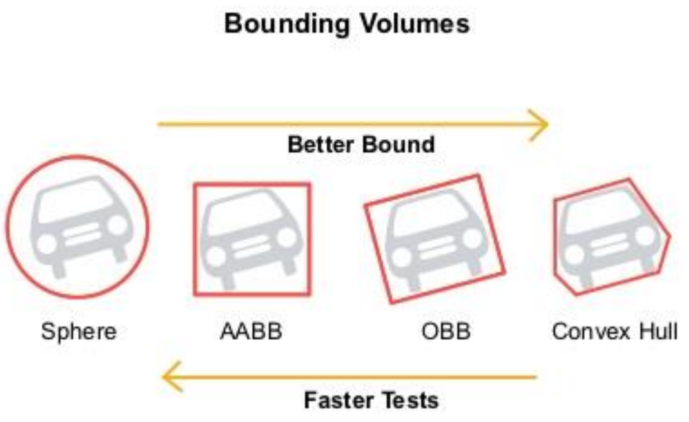

Figure 3 A comparison of a few possible collision detection algorithms (Serrano, 2016).

Axis-Aligned Bounding Box collision detection, or AABB for short, works in a way that it checks collision for two bounding boxes by comparing two points from each box, one with minimal values for the X, Y, and Z axis, and the second with maximum values.

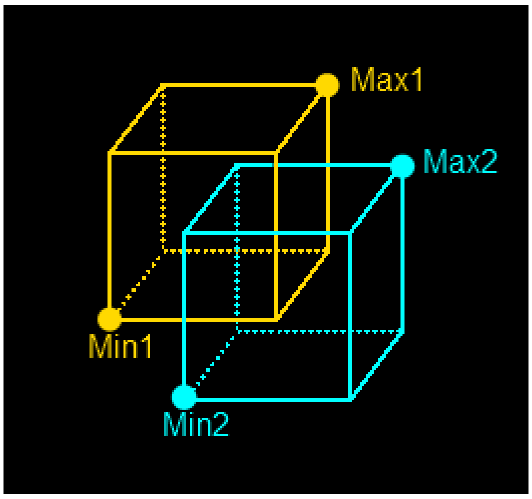

Figure 4 How AABB works (Casillas, 2010).

If the max point of the first box is greater in its values than the min point of the second box, and the min point of the first box is less in its values than the max point of the second box, then the two boxes are overlapping – therefore colliding (Casillas, 2010).

For the boxes to fall properly and not just move a certain distance each timestep, gravity is needed to be included, as well as an integration method to calculate the distance a box has travelled after the current timestep. As the project does not require realistic physics, and we are more concerned about computational time, I decided to use the Euler integration method (Dawkins, 2018).

$$v(t_0 + \Delta t) = v_0 + a\Delta t$$

The Euler method calculates the velocity after the given timestep (in the case of games, delta time) by adding the change of velocity happening in the next time step to the current velocity.

As for the lighting, I decided to use the Blinn-Phong lighting model, as it is an improvement on the Phong lighting model.

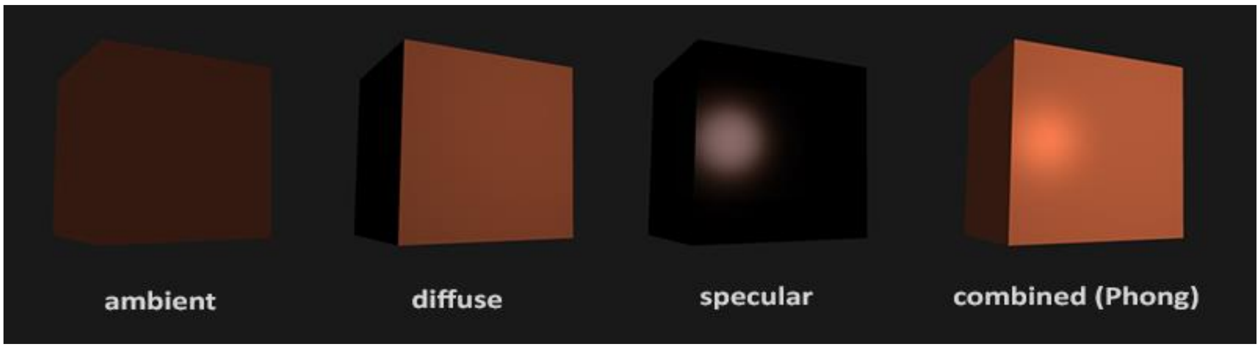

Figure 5 Phong lighting model (Vries, 2017).

The Phong model calculates a reflection vector for the specular lighting by reflecting the light direction around the normal vector, and then calculates the angle between the reflection vector and the view direction. The smaller the angle between them, the greater the impact of the specular light (Vries, 2017). This can, however, cause problems when the angle goes over 90 degrees, nullifying the specular lighting component in some cases.

In the Blinn-Phong model, instead of using a reflection vector for specular lighting, a halfway vector is created, which is a unit vector exactly halfway between the view direction and the light direction. The closer this halfway vector aligns with the surface's normal vector, the higher the specular contribution. (Vries, 2017).

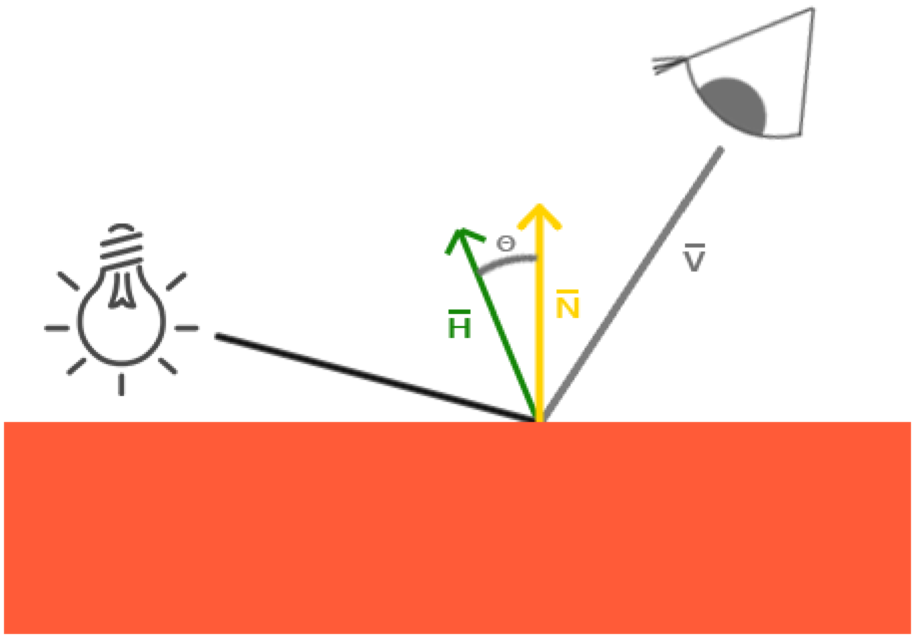

Figure 6 Specular lighting calculations for Blinn-Phong model (Vries, 2017).

As the player controls a first person camera, it rotates its display using Euler angles, which means using a spherical coordinate system.

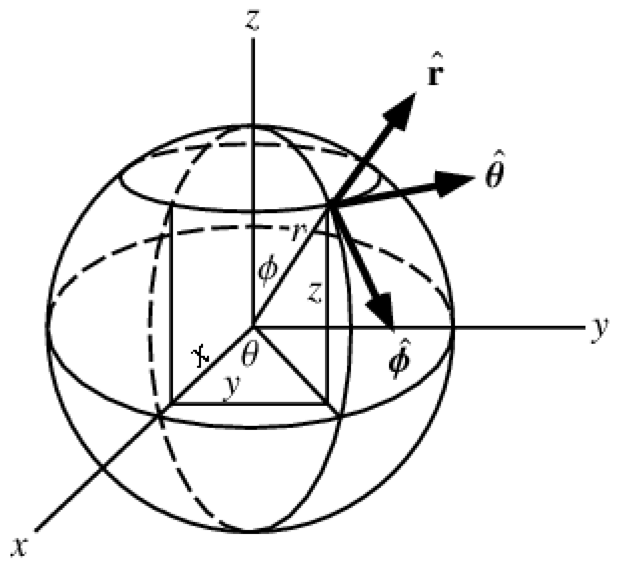

Figure 7 Spherical coordinates (Weisstein, c.a. 2018).

Therefore, we need to convert them to Cartesian coordinates with each update to correctly show the world with our camera using these equations:

Transformation coordinates

Spherical $(r,\theta,\phi) \rightarrow$ *Cartesian* (x,y,z)

$x = r\sin\phi\cos\theta$

$y = r\sin\phi\sin\theta$

$z = r\cos\phi$

Figure 8 Convertion of coordinates (Weisstein, c.a. 2018)..

Description of the program

The program starts with initializing the game, which creates a window, initializes GLEW and calls the menu() function. In this function, the program initializes a background for the menu, which is a plane, using set vec3 positions and texture coordinates, as well as a texture and shaders, which are then used by the constructors of VertexArray, Texture and ShaderProgram classes. It also initializes a box, which will rotate near the buttons. Later, it initializes the SFML buttons and text for the menu. Finally, it enters a while loop during which it renders the background, box, text and buttons, and awaits user input, which is clicking on one of the buttons – “Play the Game”, “Instructions” and “Quit the Game”. Clicking on the Instructions buttons runs the instructions() function, which practically does everything what menu does, except that the text is changed to the instructions of the game. Clicking on “Play the Game” runs the actual game, executing the gameLoop() function.

Below is an UML class diagram of the whole program.

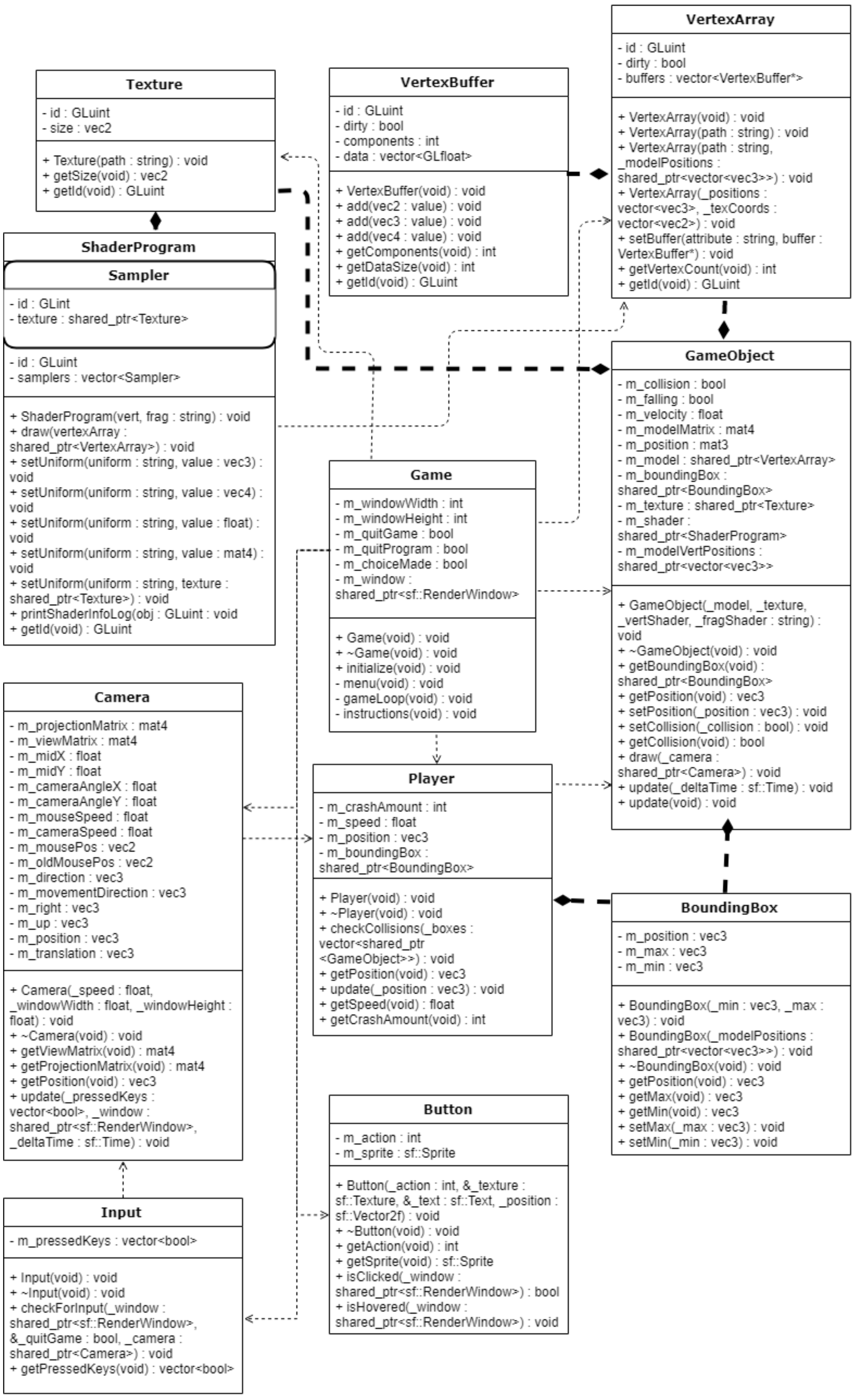
Texture
- id : GLuint
- size : vec2
+ Texture(path : string) : void
+ getSize(void) : vec2
+ getId(void) : GLuint
VertexBuffer
- id : GLuint
- dirty : bool
- components : int
- data : vector<GLfloat>
+ VertexBuffer(void) : void
+ add(vec2 : value) : void
+ add(vec3 : value) : void
+ add(vec4 : value) : void
+ getComponents(void) : int
+ getDataSize(void) : int
+ getId(void) : GLuint
VertexArray
- id : GLuint
- dirty : bool
- buffers : vector<VertexBuffer*>
+ VertexArray(void) : void
+ VertexArray(path : string) : void
+ VertexArray(path : string, _modelPositions : shared_ptr<vector<vec3>>) : void
+ VertexArray(_positions : vector<vec3>, _texCoords : vector<vec2>) : void
+ setBuffer(attribute : string, buffer : VertexBuffer*) : void
+ getVertexCount(void) : int
+ getId(void) : GLuint
ShaderProgram
Sampler
- id : GLint
- texture : shared_ptr<Texture>
- id : GLuint
- samplers : vector<Sampler>
+ ShaderProgram(vert, frag : string) : void
+ draw(vertexArray : shared_ptr<VertexArray>) : void
+ setUniform(uniform : string, value : vec3) : void
+ setUniform(uniform : string, value : vec4) : void
+ setUniform(uniform : string, value : float) : void
+ setUniform(uniform : string, value : mat4) : void
+ setUniform(uniform : string, texture : shared_ptr<Texture>) : void
+ printShaderInfoLog(obj : GLuint : void
+ getId(void) : GLuint
GameObject
- m_collision : bool
- m_falling : bool
- m_velocity : float
- m_modelMatrix : mat4
- m_position : mat3
- m_model : shared_ptr<VertexArray>
- m_boundingBox : shared_ptr<BoundingBox>
- m_texture : shared_ptr<Texture>
- m_shader : shared_ptr<ShaderProgram>
- m_modelVertPositions : shared_ptr<vector<vec3>>
+ GameObject(_model, _texture, _vertShader, _fragShader : string) : void
+ ~GameObject(void) : void
+ getBoundingBox(void) : shared_ptr<BoundingBox>
+ getPosition(void) : vec3
+ setPosition(_position : vec3) : void
+ setCollision(_collision : bool) : void
+ getCollision(void) : bool
+ draw(_camera : shared_ptr<Camera>) : void
+ update(_deltaTime : sf::Time) : void
+ update(void) : void
Game
- m_windowWidth : int
- m_windowHeight : int
- m_quitGame : bool
- m_quitProgram : bool
- m_choiceMade : bool
- m_window : shared_ptr<sf::RenderWindow>
+ Game(void) : void
+ ~Game(void) : void
+ initialize(void) : void
- menu(void) : void
- gameLoop(void) : void
- instructions(void) : void
Camera
- m_projectionMatrix : mat4
- m_viewMatrix : mat4
- m_midX : float
- m_midY : float
- m_cameraAngleX : float
- m_cameraAngleY : float
- m_mouseSpeed : float
- m_cameraSpeed : float
- m_mousePos : vec2
- m_oldMousePos : vec2
- m_direction : vec3
- m_movementDirection : vec3
- m_right : vec3
- m_up : vec3
- m_position : vec3
- m_translation : vec3
+ Camera(_speed : float, _windowWidth : float, _windowHeight : float) : void
+ ~Camera(void) : void
+ getViewMatrix(void) : mat4
+ getProjectionMatrix(void) : mat4
+ getPosition(void) : vec3
+ update(_pressedKeys : vector<bool>, _window : shared_ptr<sf::RenderWindow>, _deltaTime : sf::Time) : void
Player
- m_crashAmount : int
- m_speed : float
- m_position : vec3
- m_boundingBox : shared_ptr<BoundingBox>
+ Player(void) : void
+ ~Player(void) : void
+ checkCollisions(_boxes : vector<shared_ptr<GameObject>>) : void
+ getPosition(void) : vec3
+ update(_position : vec3) : void
+ getSpeed(void) : float
+ getCrashAmount(void) : int
BoundingBox
- m_position : vec3
- m_max : vec3
- m_min : vec3
+ BoundingBox(_min : vec3, _max : vec3) : void
+ BoundingBox(_modelPositions : shared_ptr<vector<vec3>>) : void
+ ~BoundingBox(void) : void
+ getPosition(void) : vec3
+ getMax(void) : vec3
+ getMin(void) : vec3
+ setMax(_max : vec3) : void
+ setMin(_min : vec3) : void
Button
- m_action : int
- m_sprite : sf::Sprite
+ Button(_action : int, &_texture : sf::Texture, &_text : sf::Text, _position : sf::Vector2f) : void
+ ~Button(void) : void
+ getAction(void) : int
+ getSprite(void) : sf::Sprite
+ isClicked(_window : shared_ptr<sf::RenderWindow>) : bool
+ isHovered(_window : shared_ptr<sf::RenderWindow>) : void
Input
- m_pressedKeys : vector<bool>
+ Input(void) : void
+ ~Input(void) : void
+ checkForInput(_window : shared_ptr<sf::RenderWindow>, &_quitGame : bool, _camera : shared_ptr<Camera>) : void
+ getPressedKeys(void) : vector<bool>

Figure 9 UML class diagram of the program.

I decided to put the Player into an own separate class, while the floor and crates are of the GameObject class. This is due to the fact that the floor and crates share the same uniforms for lighting in shaders to make them easily distinguishable from the dark skybox, and the only thing required to separate them was two update functions, one with the delta time required to move the crates, and one with no parameters to just update the model matrix for the floor.

Constructing a Player or GameObject object creates a BoundingBox as well, which is created from either the vertex positions of a loaded 3D model in VertexArray, or created from pre-set positions, as in case of Player. Since the player is represented by only the position of the camera, which is a single point, we therefore need to generate an imaginary bounding box around the player.

The Input class is used to store the pressed keys in a vector, as well as handle all of the SFML events such as keyboard and mouse inputs, as well as closing the window.

The Camera class is used to move the camera basing on the inputs received from the Input object, as well as returning the view and projection matrices for drawing the shapes.

The game loop therefore, after initializing the Input, Camera and Player classes, the floor object of GameObject class, the skybox, the clocks, times, sounds and HUD text, enters a while loop:

```
While window is open and game is not quitted
        Clear the window
        Disable depth testing
        Draw the skybox
        Enable depth testing
        Check for inputs with the Input object
        Update the camera's position and rotation, as well as its view matrix
        Update the player's position and its bounding box position, using the camera position
        If the fall has begun
                        If fall has not been begun yet
                                Start the "survival time" clock
                                Set the fall as begun
                        End if
                Update the survival time
                Add the delta time to the event timer
                If event timer >= 0.08 seconds
                        Increment amount of spawned boxes
                        Create a new crate (GameObject)
                        Set the position to random X and Z values located on the plane,
                        Y to 200
                        Push it back onto the crate vector

                End if
                If the crate vector is not empty
                        Update each crate's position using delta time
                        Draw each crate
                        If the crate's position is below -3.0
                Erase the crate from the vector (destroy it)
            End if
            Check if the player collided with any of the crates
            If any of the crates collided
                Play a sound
                Erase the crate from the vector (destroy it)
                If current survival time is a record
                    Set best time to current time
                End if
                Restart the crash time clock
            End if
        End if
    End if
    Update the floor
    Draw the floor
    Set the delta time
    Update the HUD with values
    Draw the HUD
    Display everything
End while
```

Analysis and conclusion

The collision detection system works fine, with the simple broad phase mechanic of checking the Y position of each crate. With the size of the player's bounding box used to deal with the rotations of the camera, situations where the player seemed to dodge a crate but was actually registered by the program to collide do not seem to be actually noticeable. A possible room for improvement to ensure that such possibility completely disappears would be to move from AABB to Oriented Bounding Boxes, however the implementation of such would be needed to be checked if it would be as computationally efficient as AABB.

The currently used HUD display is definitely easy to gain information from, and not confusing the player. A disadvantage of using SFML for drawing the GUI/HUD though is that it has to store all of the currently used OpenGL states (such as glUseProgram, glEnable etc.) before drawing the button sprites or text, and then removing all of them from the stack for the next tick/run of the game loop. As checked by removing the lines responsible for drawing the in-game text and storing the OpenGL states, there was no noticeable difference in the amount of frames per second, though my test has a critical flaw as I only made the test on a very small set of machines. For accurate results, it would be needed to check this on multiple computers with different specifications. However, it is important to remember that the lack of frames per second difference might be the case for a small scale project like this, whereas in a wide-scale projects it could make a noticeable difference. This could be improved by either implementing an OpenGL library for text, such as the OpenGL-FreeType Library, or by using a bitmap font text generator, to skip using SFML for drawing altogether.

The currently drawn skybox does not seem unreal at the first glance, which is one of the reasons why I decided to go with a space texture for the skybox. However, deliberate looking at the edges of the cube shows that the texture is repeating. A room of improvement would be to switch to a cubemap. A noticeable disadvantage is that the skybox is currently drawn by disabling the depth test and drawing it first before any objects are visible on the screen. This, however, is a bit inefficient as it runs the fragment shader for every pixel on the screen, even if the skybox is completely covered by models. This should not make much different on a small scale project like this, but it could potentially affect performance in, for example, an open-world wide-scale RPG game. Unfortunately, my optimization tries ended in either having the crates suddenly show up at a small Y position, or having the depth buffer clear the skybox with the default clear colour.

To conclude, I believe that for an initial OpenGL project, the most important objective is achieved, which is giving at least short-term enjoyment from playing the game, even though it would need tweaking for expanding the game enjoyment time if this project would be worked on in the future, such as dropping power-ups allowing to slow down the time, adding another game mechanic - shooting down the crates, or changing the camera into 3rd person mode and making the player a spaceship omitting asteroids.

Report references

Figure 3: Serrano H. (2016). *Tips for developing a Collision Detection System*. [online]. Available at: https://www.haroldserrano.com/blog/tips-for-developing-a-collision-detection-system [Accessed 18 Jan 2019].

Figure 4: Casillas M. (2010). *AABB to AABB.* [online] Available at: http://www.miguelcasillas.com/?p=30 [Accessed 18 Jan 2019].

Figure 5: Vries J. (2017). *Basic Lighting*. [online] Available at: https://learnopengl.com/Lighting/Basic-Lighting [Accessed 18 Jan 2019].

Figure 6: Vries J. (2017). *Advanced Lighting*. [online] Available at: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting [Accessed 18 Jan 2019].

Figure 7: Weisstein E. (c.a. 2018) *Spherical Coordinates.* [online] Available at: http://mathworld.wolfram.com/SphericalCoordinates.html [Accessed 18 Jan 2019].

Figure 8: Weisstein E. (c.a. 2018) *Spherical Coordinates.* [online] Available at: http://mathworld.wolfram.com/SphericalCoordinates.html [Accessed 18 Jan 2019].

Casillas M. (2010). *AABB to AABB.* [online] Available at: http://www.miguelcasillas.com/?p=30 [Accessed 18 Jan 2019].

Dawkins P. (2018). *Euler's Method*. [online] Available at: http://tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx [Accessed 18 Jan 2019].

Vries J. (2017). *Basic Lighting*. [online] Available at: https://learnopengl.com/Lighting/Basic-Lighting [Accessed 18 Jan 2019].

Vries J. (2017). *Advanced Lighting*. [online] Available at: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting [Accessed 18 Jan 2019].

Weisstein E. (c.a. 2018) *Spherical Coordinates.* [online] Available at: http://mathworld.wolfram.com/SphericalCoordinates.html [Accessed 18 Jan 2019].

Used materials

Menu background: texturify.com (2016). *Sundown sky*. [online] Available at: http://texturify.com/stock-photo/sundown-sky-10774.html [Accessed 18 Jan 2019].

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-canvas-peel-and-stick-wall-mural-dt3501.html#product-details-tab-specification [Accessed 18 Jan 2019].

Button texture: graphicburger.com (2017). *6 Marble Textures Vol.3*. [online] Available at: https://graphicburger.com/6-marble-textures-vol-3/ [Accessed 18 Jan 2019].

Skybox texture: wallsfield.com (2015). *Stars Texture Free HD Wallpapers*. [online] Available at: https://wallsfield.com/stars-texture-hd-wallpapers/ [Accessed 18 Jan 2019].

Crate model and texture: marcis3D (2013). *Wooden Crate*. [online] Available at: https://www.turbosquid.com/3d-models/free-wooden-crate-3d-model/732618 [Accessed 18 Jan 2019].

Crate hit sound: dxeyes (2018). *crate_break_3.wav*. [online] Available at: https://freesound.org/people/dxeyes/sounds/432669/ [Accessed 18 Jan 2019].

Music: Sirius Beat (2013). *One*. [online] Available at: https://www.youtube.com/watch?v=foKOqaimVv4 [Accessed 18 Jan 2019].