
James Paton
S1111175
Computer Games (Software Development)

# Games Programming 2
# (M3I622943)
# Coursework Documentation

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.*

*James Paton*

December 2018

Contents

# 1. Entry Point

## 1.1. main

### Functions

`int main()`
Creates a `MainGame` called `mainGame` and calls its `Run` function. Once the program terminates, the function returns 0.

## 1.2. Main Game

### Class Variables

**Private**
`Display _gameDisplay`
A `Display` is used to setup and store the SDL Window and the SDL Context.

`GameState _gameState`
An enumerated type which stores the current state of the game.

`GameObjectManager* goMan = GameObjectManager::GetInstance()`
Gets a pointer to the game object manager.

`InputManager* input = InputManager::GetInstance()`
Gets a pointer to the input manager.

`AudioManager* audioMan = AudioManager::GetInstance()`
Gets a pointer to the audio manager.

`CollisionManager* colMan = CollisionManager::GetInstance()`
Gets a pointer to the collision manager.

`Camera* camera1 = goMan->Instantiate<Camera>()`
Instantiates a `Camera` game object named `camera1`.

`Camera* camera2 = goMan->Instantiate<Camera>()`
Instantiates a `Camera` game object named `camera2`.

`Light* light = goMan->Instantiate<Light>()`
Instantiates a `Light` game object named `light`.

```
Monkey* monkey1 = goMan->Instantiate<Monkey>("..//Resources//Bricks.jpg")
Monkey* monkey2 = goMan->Instantiate<Monkey>("..//Resources//Water.jpg")
Monkey* monkey3 = goMan->Instantiate<Monkey>("..//Resources//notexture.jpg")
```
Instantiates `Monkey` game objects named `monkey1`, `monkey2`, and `monkey3`. The parameter is passed through the game object manager to the `Monkey` constructor and is used to set the texture of the monkey.

`Terrain* terrain = goMan->Instantiate<Terrain>()`
Instantiates a `Terrain` game object named `terrain`.

### Functions

**Private**
`void InitSystems()`
Initialises `_gameDisplay` with `initDisplay()`. The background music (`Background.wav`) is loaded into the audio manger and named `BackgroundMusic`, this track is then played with `playSound("BackgroundMusic")`. The two cameras in the scene have their `Init()` functions called, defining the FOV, the display aspect ratio, and the near and far clipping planes. These cameras are also repositioned using their attached `Transform` components. Camera 2 has its tag set to the empty string and its `UserControl` component disabled. The three monkeys have their positions defined and the third monkey has a `Demo` component attached. The light is positioned and its `UserControl` component is disabled.

`void MainLoop()`
Main loop contains a while loop that will only exit once `_gameState` is not equal to `GameState::EXIT`. Inside the loop: `Time` is updated; the displayed is cleared; the collision, input, and game object managers are updated; `ToggleControl()` is called; and the buffers of the display are swapped.

`void ToggleControl()`
Toggles control swaps which camera is viewed from and controlled by the user when C is pressed. If right click is held, the user has control over the light instead.

**Public**
`MainGame()`
Sets `_gameState` to `GameState::PLAY` and sets `_gameDisplay` equal to a pointer to a new `Display`.

`~MainGame()`
Empty destructor.

`void Run()`
Calls `InitSystems()` and `MainLoop()`.

# 2. Managers
## 2.1. Game Object Manager

### Class Variables

**Private**
`static GameObjectManager* instance`
Pointer to `GameObjectManager` named `instance`.

`std::list<GameObject*> gameObjects`
List of `GameObject` pointers managed.

### Functions

**Private**
`GameObjectManager()`
Private constructor to prevent instancing, as `GameObjectManager` is a singleton.

**Public**
`~GameObjectManager()`
Empty destructor.

`static GameObjectManager* GetInstance()`
If no `GameObjectManager` exists, one will be created on the heap and a pointer to it return. If one already exists, the pointer to it will be returned.

`void Delete(GameObject* go)`
The `GameObject go` has its variable `toDelete` set true, objects marked as such are handled differently in `Update()`.

`void Update()`
This function loops through all `GameObject` stored in `gameObjects` that are not set to be deleted and calls their `UpdateAll()`. After this, any game objects are deleted, if required.

`T* Instantiate()`
This is a generic function that will create a game object of type `T`. The use of "`static_assert(std::is_base_of<GameObject, T>::value, "T must derive from GameObject")`" [ILDJARN, 2011] ensures that `T` inherits from `GameObject`. If the type does inherit from game object, the object is created on the heap, a pointer to it is added to `gameObjects`, its `Awake()` function is called, and then the pointer is returned.

`T* Instantiate(T2 parameter)`
This overload function is similar to `T* Instantiate()` above expect this one allows the creation of game objects with one parameter in their constructors.

`T* GetGameObjectTag(std::string tag)`
This function returns a game object tagged with `tag`. If no game object can be found, an error is thrown.

## 2.2. Asset Manager

### Class Variables

**Private**
`static AssetManager* instance`
Static pointer to `AssetManager` named `instance`.

**Public**
`std::map<std::string, GLuint> vaos`
`std::map<std::string, unsigned int> drawCounts`
Two maps, one used to store the `GLuint` of vertex array objects, and one to store draw counts. These are used inside `Mesh` to prevent the reloading of any meshes that have already been loaded from file. The string stored is the file directory as this is guaranteed to be unique.

`std::map<std::string, GLuint> textures`
A map used to store the `GLuint` of textures. This is used inside `Texture`, like above.

`std::map<std::string, unsigned int> sounds`
A map used to store the `unsigned int` of sounds. This is used inside `AudioManager`, like above except the string refers to a name given to the sound when loaded.

### Functions

**Private**
`AssetManager()`
Private constructor to prevent instancing, as `AssetManager` is a singleton.

**Public**
`static AssetManager* GetInstance()`
If no `AssetManager` exists, one will be created on the heap and a pointer to it return. If one already exists, the pointer to it will be returned.

`~AssetManager()`
Empty destructor.

## 2.3. Collision Manager

### Class Variables

**Private**
`static CollisionManager* instance`
Static pointer to `CollisionManager` named `instance`.

`std::list<Collider*> colliders`
`std::list<Collision*> collisions`
Lists of all the colliders attached to game objects and all the collisions currently taking place.

### Functions

**Private**
`CollisionManager()`
Private constructor to prevent instancing, as `CollisionManager` is a singleton.

`bool CheckCollision(Sphere* sphere1, Sphere* sphere2)`
Checks whether two `Sphere` are colliding.

**Public**
`static CollisionManager* GetInstance()`
If no `CollisionManager` exists, one will be created on the heap and a pointer to it return. If one already exists, the pointer to it will be returned.

`~CollisionManager()`
Empty destructor.

`void Add(Collider* collider)`
`void Remove(Collider* collider)`
Allows colliders to be added and removed from the collision manager.

`void Update()`
This functions first clears the list of collisions. It then loops through all combinations of colliders (without duplicates) and creates a collision for every two game objects which are currently colliding.

`bool Check(GameObject* go)`
Used to check whether a given game object is colliding with anything.

### Structs

`Collision`
Contains pointers to two game objects that have collided.

## 2.4. Input Manager

### Class Variables

**Private**
`static InputManager* instance`
Static pointer to `InputManager`.

`std::list<SDL_Keycode> keys`
`std::list<SDL_Keycode> keysDown`
`std::list<SDL_Keycode> keysToggle`
Lists of all keys currently being held down, those pressed this frame, and currently toggled on.

`float mouseX, mouseY`
Coordinates of cursor on window.

`std::list<unsigned int> mouseButtons`
`std::list<unsigned int> mouseButtonsDown`
Lists of all mouse buttons currently being held down and those pressed this frame.

`bool cursorLocked`
Whether the cursor is locked to the centre of the window and hidden.

### Functions

**Private**
`InputManager()`
Private constructor to prevent instancing, as `InputManager` is a singleton.

`void AddKey(SDL_Keycode key)`
`void RemoveKey(SDL_Keycode key)`
Correctly add and remove keys from the lists `keys`, `keysDown` and `keysToggle`.

`void LockCursor(bool lock)`
Lock or unlock the cursor.

**Public**
`~InputManager()`
Empty destructor.

`static InputManager* GetInstance()`
If no `InputManager` exists, one will be created on the heap and a pointer to it return. If one already exists, the pointer to it will be returned.

`void Update(Display* display, GameState* gameState)`
This function begins by clearing data from the previous frame: setting the mouse coordinates to zero, and clearing the two lists `keysDown` and `mouseButtonsDown`. The function then loops through all events with `while (SDL_PollEvent(&evnt))`. If the even is an `SDL_QUIT` the `GameState` is changed to `EXIT` and the program will exit at the end of this game loop. The rest of the events are handled in a switch statement. Events such as `SDL_MOUSEMOTION`, which updates the mouse coordinates; `SDL_MOUSEBUTTONDOWN` and `SDL_MOUSEBUTTONUP`, which updates the lists of mouse buttons pressed; and finally, `SDL_KEYDOWN` and `SDL_KEYUP`, which updates the lists of keys pressed. The last function called is `SDL_WarpMouseInWindow`, if the cursor is currently locked, this function will keep the cursor in the centre of the window.

`bool GetKey(SDL_Keycode key)`
`bool GetKeyDown(SDL_Keycode key)`
`bool GetKeyToggle(SDL_Keycode key)`
Check the corresponding lists if `key` exists.

`float GetMouseX()`
`float GetMouseY()`
Getters for the mouse coordinates

`bool GetMouseButton(unsigned int button)`
`bool GetMouseButtonDown(unsigned int button)`
Check the corresponding lists if `button` exists.

# 3. Game Objects

## 3.1. Game Object

### Class Variables

**Private**
`std::list<Component*> components`
A list of `Component` pointers attached to this game object.

`std::list<Component*> componentsStart`
A list of components whose `Start()` functions are yet to be called.

`bool startCalled`
A boolean used to keep track of whether or not a game object's `Start()` has been called.

**Public**
`std::string tag = "untagged"`
The game object's tag.

`bool toDelete = false`
A boolean used to keep track of whether a game object should be deleted.

### Functions

**Private**
`void UpdateGameObject()`
A function used to called the game object's `Start()` and `Update()` functions. `Start()` is only called once, `Update()` every game loop.

`void UpdateComponents()`
Calls the game object's component's `Start()` and `Update()` functions. These functions are only called if that component is enabled and, as with `GameObject`, `Start()` is only called once, `Update()` every game loop.

**Public**
`GameObject()`
Empty constructor.

`virtual ~GameObject()`
Destructor which deletes all of the game object's components. It is virtual so that inherited game object's destructors are also called.

```
virtual void Awake() = 0
virtual void Start() = 0
virtual void Update() = 0
```
Pure virtual functions which must be defined when inheriting from `GameObject`.

`void AddComponent(Component* c)`
A function for adding a `Component` to the game object. The components type is stored in `type` using `typeid()` from `<typeinfo>`. The component is added to the `components` and `componentsStart` lists. Finally, the components `gameObject` is updated with a pointer to this game object.

`void RemoveComponent(Component* c)`
For each component attached to this game object, if the component's `type` matches the type of `c`, it is removed from the `components` and `componentsStart` lists and then deleted. NB: This function does not work correctly if a game object has more than one of the same type of component attached. If this happens, all will be removed, but only the final one will be deleted.

`T* GetComponent()`
This function returns a pointer to the attached component of type `T`. If no component of type `T` is found, an error is thrown.

`void UpdateAll()`
This function calls both `UpdateGameObject()` and `UpdateComponents()`.

## 3.2. Camera

### Functions

**Public**
`Camera()`
`~Camera()`
Empty constructor and destructor.

`void Awake()`
This function is called as soon as the game object is initialised. Here, the camera is tagged `"MainCamera"`, allowing it to be found by `Shader`. The components which constitute a camera are also added: `Transform`, `CameraComponent`, and `UserControl`.

`void Start()`
`void Update()`
Empty functions required by `GameObject`.

### 3.3. Monkey

#### Class Variables

**Private**
`std::string texName`
A string holding the location of the texture require for this monkey.

#### Functions

**Public**
`Monkey(std::string name)`
Constructor for `Monkey`, requiring a texture to be specified which is assigned to `texName`.

`~Monkey()`
Empty destructor.

`void Awake()`
A function which is called as soon as a game object is initialised. In this case, it is used to add all the components which constitute a monkey: `Transform`, `Mesh`, `Texture`, `Shader`, and a `Collider` of radius `1.1f`.

`void Start()`
This function is called only once, this is where several of the monkey's components are initialised: which shader, mesh, and texture required to render the monkey are loaded.

`void Update()`
This function is called every game loop and is it where the monkey is rendered. To render a monkey its shader must be bound, its shader updated, its texture bound, and finally the mesh's `draw()` function is called.

## 4. Components

### 4.1. Component

#### Class Variables

**Public**
`std::string type`
A string which holds the type of the inherited component, as defined by `typeid()`.

`GameObject* gameObject`
A pointer to this component's game object.

`bool enabled = true`
A boolean for whether this component is enabled.

#### Functions

**Public**
`Component()`
Empty constructor.

`virtual ~Component()`
Empty virtual destructor. It is virtual so that the destructors of inherited components are also called.

`virtual void Start() = 0`
`virtual void Update() = 0`
Pure virtual functions which must be defined when inheriting from `Component`.

## 4.2. Camera Component

### Class Variables

**Private**
`glm::mat4 projection`
This variable holds the projection matrix (the frustum) of the camera.

### Functions

**Public**
`CameraComponent()`
`~CameraComponent()`
Empty constructor and destructor.

`void Start()`
`void Update()`
Empty functions required by `Component`.

`void Init(float fov, float aspect, float nearClip, float farClip)`
This function is used to initialize the camera, it defines the FOV, aspect ratio, and the near and far clipping planes of the camera which are required to create the projection matrix.

`glm::mat4 GetViewProjection()`
This function returns a `glm::mat4` which is the view-projection matrix of this camera. This matrix is created by multiplying the projection matrix by the view matrix. A function, `glm::lookAt()`, is used to created the view matrix from information stored in this object's `Transform`.

## 4.3. Collider

### Class Variables

**Public**
`CollisionManager* colMan`
A pointer to the collision manager.

`Sphere* sphere = new Sphere()`
The sphere that represents this collider's collidable area.

### Functions

**Public**
`Collider(float radius)`
Constructor which updates `sphere` with `radius` and adds itself to the collision manager.

`~Collider()`
Destructor which removes this collider form the collision manger.

`void Start()`
Empty function required by `Component`.

`void Update()`
Updates this sphere's position with the position of this game object's `Transform`.

### Structs

`Sphere`
A struct which defines a sphere: a `Vector3 position` and `float radius`. It also contains getters and setters for those variables.

## 4.4. Demo

### Class Variables

**Public**
`GameObjectManager* goMan = GameObjectManager::GetInstance()`
`CollisionManager* colMan = CollisionManager::GetInstance()`
Get pointers to the game object manager and the collision manager.

`Monkey* monkey1`
`Monkey* monkey2`
Define two pointers to two monkeys.

### Functions

**Public**
`Demo()`
`~Demo()`
Empty constructor and destructor.

`void Start()`
This function is called once at the start of the next game loop. It calls `MakeMonkey1()`.

`void Update()`
This function first gets the `Transform` of the game object that this component is attached to, in this case, it is the missing texture monkey (`monkey3`) from MainGame. The object's position is updated following an elliptical pattern designed to cause it to intersect with the other four monkeys in the scene (`monkey1` and `monkey2` from MainGame, as well as `monkey1` and `monkey2` from Demo). An intersection with `monkey1` from Demo will cause it to be deleted and `monkey2` instantiated. The opposite is true for when this object intersects with `monkey2`.

`void MakeMonkey1()`
`void MakeMonkey2()`
These functions each instantiate a monkey (either `monkey1` or `monkey2`), one with the carpet texture, and one with the metal texture. The monkeys are also instantiate in different locations.

## 4.5. Transform

### Class Variables

**Private**
`Vector3 pos`
`Vector3 scale`
`Vector3 eulerAngles`
Three `Vector3` containing information about the game objects position, scale and angle in radian Euler angles.

`Quaternion quaternion`
The game object's rotation stored as a quaternion.

### Functions

**Public**
`Transform()`
`Transform(Vector3 pos, Vector3 scale, Vector3 rot)`
`Transform(Vector3 pos, Vector3 scale, Quaternion quaternion)`
`Transform` constructors. Default constructor initialises `pos` and `eulerAngles` to a `Vector3` of (0, 0, 0), `scale` to (1, 1, 1) and the quaternion to the default quaternion (1, 0, 0, 0). Overloaded constructors are provided for specifying a game object's initial position, scale, and rotation in either degree Euler angles or as a quaternion.

`void Start()`
`void Update()`
Empty functions required to be defined as they are pure virtual in `Component`.

`glm::mat4 GetModelMatrix()`
Returns the model matrix of this transform. A matrix form of `pos`, `scale`, and `quaternion` is fetched and then multiplied together to be return as a `glm::mat4` model matrix.

`Vector3 GetPos()`
`Vector3 GetScale()`
`Vector3 GetEulerAngles()`
`Quaternion GetRotation()`
Getters for the transform's position, scale, and rotation as either degree Euler or as a quaternion.

`void SetPos(Vector3 pos)`
`void SetScale(Vector3 scale)`
`void SetRotate(Vector3 eulerAngles)`
`void SetRotate(Quaternion q)`
Setters for the transform's position, scale, and rotation as either degree Euler or as a quaternion. Passing in Euler will also update the quaternion and vice versa.

`void TranslateLocal(Vector3 pos)`
`void TranslateGlobal(Vector3 pos)`
These functions allow the game object to be translated on either the global axes or on the object's local axes. The global translation is a simple addition of the current position with `pos`. The local transform is similar except `pos` must first be transformed by the object's rotation matrix.

`void RotateLocal(Vector3 eulerAngles)`
`void RotateGlobal(Vector3 eulerAngles)`
These functions allow the game object to be rotated on either the global axes or on the object's local axes. First `eulerAngles` is converted to radians and then to a quaternion. To perform the rotation locally, the object's current quaternion and the new quaternion are multiplied together with the object's current quaternion on the left. For a global rotation, the same operation is applied expect the object's quaternion is on the right. The game object's Euler angles are then updated.

`Vector3 GetForward()`
`Vector3 GetUp()`
`Vector3 GetRight()`
These functions are used to get a direction vector of the game object's forward, up, or right. These vector are the result of the multiplication of the object's rotation matrix and the equivalent global direction vector.

## 4.6. User Control

### Functions

**Public**
`UserControl()`
Empty constructor.

`~UserControl()`
Empty destructor

`void Start()`
Empty start function required by `Component`.
`void Update()`
This function is called every game loop and allows the game object that this component is attached to to be controlled with mouse and keyboard input. The function begins by defining the movement speed (`speed`) and rotation speed (`rotationSpeed`), these values are multiplied by `Time::deltaTime` to ensure the objects movement speed in consistent as frame rates fluctuate. Then, two pointers are retrieved: `InputManager* input = InputManager::GetInstance()` which allows mouse and keyboard input to be queried, and `Transform* transform = gameObject->GetComponent<Transform>()` which gets the transform of the same game object that this component is attached to. These pointers allow if statements regarding key presses to result in the movement and rotation of the game object. Mouse movement is also translated into rotation of the game object.

# 5. Other Scripts

## 5.1. Enums

### Class Variables

**Public**
`enum class GameState { PLAY, EXIT }`
Defines an enumerated type regarding the state of the game.

## 5.2. Quaternion

Much of the code for quaternions was built upon [WIKIPEDIA, 2018].

### Class Variables

**Public**
`float w, x, y, z`
These floats contain the rotation of the quaternion.

### Functions

**Public**
`Quaternion()`
`Quaternion(float pitch, float roll, float yaw)`
Constructors for `Quaternion`. The default constructor initialises `w, x, y, z` to (1, 0, 0, 0). The overloaded constructor takes pitch, roll, and yaw rotations in radians and produces a quaternion of the equivalent rotation.

`~Quaternion()`
Empty destructor.

`static glm::mat4 toRotationMatrix(Quaternion q)`
A static function for returning the 4x4 rotation matrix of a quaternion.

`static Vector3 toEulerAngle(Quaternion q)`
A static function for converting a quaternion to radian Euler angles.

`static Quaternion toQuaternion(float pitch, float roll, float yaw)`
A static function for creating a quaternion from radian Euler angles.

`static Quaternion toQuaternion(Vector3 eulerAngles)`
An overload for creating a quaternion from a `Vector3` of radian Euler angles.

`void operator=(Quaternion q)`
`bool operator==(Quaternion q)`
`Quaternion operator*(Quaternion q)`
Standard operator overloads

## 5.3. Time

### Class Variables

**Private**
`static std::chrono::high_resolution_clock timer`
This is the timer used to accurately get the current time.

`static std::chrono::steady_clock::time_point start`
`static std::chrono::steady_clock::time_point end`
These time points are used to keep track of the time at the beginning and end of `Update()`.

**Public**
`static float deltaTime`
`static float time`
`deltaTime` is the time in seconds between `Update()` calls, the time taken to compute the previous game loop. `time` is the time in seconds since the program started.

### Functions

**Private**
`Time()`
Private constructor to prevent instancing. `Time` is a pure static class.

**Public**
`static void Update()`
The current `deltaTime` is calculated by subtracting `start` from `end`. `time` is calculated by adding `deltaTime` to itself.

## 5.4. Vector3

### Class Variables

**Public**
`float x, y, z`
The three floats that define a `Vector3`.

### Functions

**Private**
`glm::mat4 RotX(float x)`
`glm::mat4 RotY(float y)`
`glm::mat4 RotZ(float z)`
These functions return the 4x4 rotation matrices for each of the x, y, and z values.

**Public**
`Vector3();`
`Vector3(float x, float y, float z)`
Constructors for the `Vector3`. The default constructor will initialize the class variables `x`, `y`, and `z` to zero. The constructor overload allows these values to be specified with the `x`, `y`, and `z` parameters.

`~Vector3()`
Empty destructor.

`glm::mat4 TransMat()`
`glm::mat4 ScaleMat()`
`glm::mat4 RotMat()`
These functions transform a `Vector3` into `glm::mat4` translation, scale, or rotation matrix.

`void Normalize()`
`Vector3 Normalized()`
`Normalize()` normalizes this `Vector3`. `Normalized()` returns a normalized version of this `Vector3` while leaving the original unchanged.

`Vector3 toRadians()`
`Vector3 toDegrees()`
Functions for converting to and from radians and degrees.

```
void operator=(Vector3 v);
bool operator==(Vector3 v);
Vector3 operator+(Vector3 v)
Vector3 operator-(Vector3 v)
Vector3 operator*(Vector3 v)
Vector3 operator/(Vector3 v)
void operator+=(Vector3 v)
void operator-=(Vector3 v)
void operator*=(Vector3 v)
void operator/=(Vector3 v)
```
Standard operator overloads.

`glm::vec3 GLM();`
Returns a `glm::vec3` of this `Vector3` with the correct values.

# 6. Shaders

The shaders below were created with help from a series of tutorials [LEARN OPENGL, 2014].

## 6.1. Lighting.vert

### Variables

**Layout**
```
layout (location = 0) in vec3 position
layout (location = 1) in vec2 texCoord
layout (location = 2) in vec3 normal
```
These variables contain the data for each vertex: the position, texture coordinates, and the normal. The value assigned to `location` corresponds to the first parameter of `glVertexAttribPointer()` and refers to each vertex array buffer created in `Mesh`.

**Out**
```
out vec3 FragPos
out vec3 Normal
out vec2 TexCoord
```
These three variables are passed out of the shader. The fragment positions, the texture coordinates, and the normals. All of which are required in the Lighting.frag shader.

**Uniform**
```
uniform mat4 vp
uniform mat4 m
```
These uniforms contain the view-projection and the model matrix.

### Functions

```
void main()
```
`gl_Position` is set to the model-view-projection matrix multiplied by the position of the vertex. `FragPos` is set to a `vec3` of the model matrix multiplied by the position of the vertex, this is so fragment positions are also rotated as the model is rotated. `normal` and `texCoord` are passed through as `Normal` and `TexCoord`.

## 6.2. Lighting.frag

### Variables

**In**
```
in vec3 FragPos
in vec3 Normal
in vec2 TexCoord
```
These three variables are passed in from the Lighting.vert shader.

**Out**
```
out vec4 FragColor
```
The result of this final shader is passed out as `FragColor` to the frame buffer.

**Uniform**
```
uniform vec3 lightColor
uniform vec3 lightPosition
```
Two uniforms are passed into this shader. The light's position and its colour.

```
uniform sampler2D sampler
```
A sampler used to get the texture colour.

### Functions

```
void main()
```
This shader calculates two different lightings: ambient and diffuse. This results of these calculations are then combined with the texture colour to light the fragment.

## 6.3. Light.vert

### Variables

**Layout**
`layout (location = 0) in vec3 position`
`layout (location = 1) in vec2 texCoord`
These variables contain the data for each vertex: the position, and texture coordinates. Normals are not required to draw the light source. The value assigned to `location` corresponds to the first parameter of `glVertexAttribPointer()` and refers to each vertex array buffer created in `Mesh`.

**Out**
`out vec2 TexCoord`
The variable containing the texture coordinates which is passed out of the shader. This information is required in the Light.frag shader.

**Uniform**
`uniform mat4 vp`
`uniform mat4 m`
These uniforms contain the view-projection and the model matrix.

### Functions

`void main()`
`gl_Position` is set to the model-view-projection matrix multiplied by the position of the vertex. `texCoord` is passed through as `TexCoord`.

## 6.4. Light.frag

### Variables

**In**
`in vec2 TexCoord`
This variable contains the texture coordinates and is passed in from the Light.vert shader.

**Out**
`out vec4 FragColor`
The result of this final shader is passed out as `FragColor` to the frame buffer.

**Uniform**
`uniform sampler2D sampler`
A sampler used to get the texture colour.

### Functions

`void main()`
The `FragColor` is assigned to `vec4(1.1, 1.1, 1.1, 1.0)` multiplied by the texture colour. This has the effect of over exposing the texture making it appear quite bright.

# 7. Missing Scripts

The following is a list of scripts included in this project but were omitted from documentation due to time constraints and document length limitations.

## Managers

- Audio Manger

## Game Objects

- Light
- Terrain

## Components

- Mesh
- Shader
- Texture

## Other Scripts

- Display
- OBJ Loader [BLY7, 2018]
- STB Image [NOTHINGS, 2018]

# 8. References


ILDJARN, 2011. *c++ - How to ensure that the template parameter is a subtype of a desired type? - Stack Overflow* [online]. Stack Overflow. [viewed 26 November 2018]. Available from: https://stackoverflow.com/questions/7020292/how-to-ensure-that-the-template-parameter-is-a-subtype-of-a-desired-type

WIKIPEDIA, 2018. *Conversion between quaternions and Euler angles – Wikipedia* [online]. Wikipedia. [viewed 12 November 2018]. Available from: https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

LEARN OPENGL, 2014. *LearnOpenGL – Basic Lighting* [online]. Learn OpenGL. [viewed 29 October 2018]. Available from: https://learnopengl.com/Lighting/Basic-Lighting

BLY7, 2018. *Bly7/OBJ-Loader: A C++ OBJ Model Loader that will parse .obj & .mtl Files into Indices, Vertices, Materials, and Mesh Structures.* [online]. GitHub. [viewed 1 October 2018]. Available from: https://github.com/Bly7/OBJ-Loader

NOTHINGS, 2018. *nothings/stb: stb single-file public domain libraries for C/C++* [online]. GitHub. [viewed 1 October 2018]. Available from: https://github.com/nothings/stb