Ben Ivory (S1621251)

**Games Programming 2**

(M31622943)

*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*.

*Signature*. Ben Ivory

Contents

[1.0 Overview of Classes and Methods 5](#_Toc533097333)

[**1.1** **main** 5](#_Toc533097334)

[1.1.1 main() 5](#_Toc533097335)

[**1.2** **MainGame** 5](#_Toc533097336)

[1.2.1 Variable 5](#_Toc533097337)

[1.2.2 MainGame() 5](#_Toc533097338)

[1.2.3 run() 5](#_Toc533097339)

[1.2.4 initSystems() 5](#_Toc533097340)

[1.2.5 gameLoop() 5](#_Toc533097341)

[1.2.6 Process Input() 5](#_Toc533097342)

[1.2.7 drawGame() 6](#_Toc533097343)

[1.2.8 AddKey() 6](#_Toc533097344)

[1.2.9 RemoveKey() 7](#_Toc533097345)

[1.2.10 CheckKey() 7](#_Toc533097346)

[1.2.11 CheckCollison() 7](#_Toc533097347)

[**1.3** **Mesh** 8](#_Toc533097348)

[1.3.1 Variables 8](#_Toc533097349)

[1.3.1.1 Vertex Variables 8](#_Toc533097350)

[1.3.1.2 BoundingSphere Variables 8](#_Toc533097351)

[1.3.1.3 Mesh Variables 8](#_Toc533097352)

[1.3.2 initModel() 8](#_Toc533097353)

[1.3.3 UpdateSphereData 9](#_Toc533097354)

[1.3.4 GetSpherePos() 9](#_Toc533097355)

[1.3.5 GetSphereRadius() 9](#_Toc533097356)

[1.3.6 mesh() 9](#_Toc533097357)

[1.3.7 ~mesh() 9](#_Toc533097358)

[1.3.8 loadModel() 9](#_Toc533097359)

[1.3.9 draw() 9](#_Toc533097360)

[**1.4** **Dispay** 10](#_Toc533097361)

[1.4.1 Variables 10](#_Toc533097362)

[1.4.2 Display() 10](#_Toc533097363)

[1.4.3 ~Display() 10](#_Toc533097364)

[1.4.4 returnError() 10](#_Toc533097365)

[1.4.5 getWidth() 10](#_Toc533097366)

[1.4.6 getHeight() 10](#_Toc533097367)

[1.4.7 swapBuffer() 10](#_Toc533097368)

[1.4.8 clearDisplay() 10](#_Toc533097369)

[1.4.9 initDisplay() 10](#_Toc533097370)

[**1.5** **Shader** 11](#_Toc533097371)

[1.5.1 Variables 11](#_Toc533097372)

[1.5.2 init() 11](#_Toc533097373)

[1.5.3 ~Shader() 11](#_Toc533097374)

[1.5.4 Bind() 11](#_Toc533097375)

[1.5.5 Update() 11](#_Toc533097376)

[1.5.6 CreateShader() 12](#_Toc533097377)

[1.5.7 LoadShader() 12](#_Toc533097378)

[1.5.8 CheckShaderError() 12](#_Toc533097379)

[**1.6** **Audio** 13](#_Toc533097380)

[1.6.1 Variables 13](#_Toc533097381)

[1.6.2 Audio() 13](#_Toc533097382)

[1.6.3 ~Audio() 13](#_Toc533097383)

[1.6.4 isBigEndian() 13](#_Toc533097384)

[1.6.5 convertToInt() 13](#_Toc533097385)

[1.6.6 LoadWAV() 13](#_Toc533097386)

[1.6.7 loadSound() 13](#_Toc533097387)

[1.6.8 playSound() 13](#_Toc533097388)

[1.6.9 playSound()b 14](#_Toc533097389)

[1.6.10 stopSound() 14](#_Toc533097390)

[1.6.11 setListener() 14](#_Toc533097391)

[**1.7** **Texture** 14](#_Toc533097392)

[1.7.1 Variables 14](#_Toc533097393)

[1.7.2 Init() 14](#_Toc533097394)

[1.7.3 ~Texture() 15](#_Toc533097395)

[1.7.4 Bind() 15](#_Toc533097396)

[**2.0** **Overview of pass through Shaders** 15](#_Toc533097397)

[**2.1** **Lighting.vert** 15](#_Toc533097398)

[2.1.1 Layouts 15](#_Toc533097399)

[2.1.2 Uniforms 15](#_Toc533097400)

[2.1.3 Variables - Out 15](#_Toc533097401)

[2.1.4 Vertex Shader 15](#_Toc533097402)

[**2.2** **Lighting.frag** 16](#_Toc533097403)

[2.2.1 Uniforms 16](#_Toc533097404)

[2.2.2 Variables – In 16](#_Toc533097405)

[2.2.3 Variables – Out 16](#_Toc533097406)

[2.2.4 Fragment Shader 16](#_Toc533097407)

[**3.0** **References** 18](#_Toc533097408)

# 1.0 Overview of Classes and Methods

* 1. **main** 
     1. main()

“main” is the entrance point for the compiler and runs “MainGame”.

* 1. **MainGame**
     1. Variable

A Display “\_gameDisplay” that sets up the SDL Window and the GL Context.

A GameState “\_gameState” that holds the current game state.

4 Mesh “mesh#” that holds data regarding a model and its bounding sphere.

A MyCamera “myCamera” that sets up the camera.

3 Texture “texture#” that hold data regarding a texture.

A Shader “shader” that handles the creatation and use of shaders.

A Audio “Audio” that handles management of the audio.

A glm::vec3 “cameraPos” that holds the position of the camera.

A glm::vec3 “lightPos” that holds the position of the lights.

A std::vector of SDL\_Keycodes “pressedKeys” to hold keyboard inputs.

A bool “CheckCollison” that determines if bounding spheres are intersecting.

* + 1. MainGame()

When called the \_gameState is updated to hold the new GameState, “PLAY” and a pointer to a new display, “\_gameDisplay” is created.

* + 1. run()

This method calls the methods “initSystems()” and “gameLoop()”.

* + 1. initSystems()

The display is initialised and “cameraPos” and “lightPos” are given values. All the “mesh#” and “texture#” are loaded from the res folder. The shader and camera are both initialised with the relevant information (Camera gets the Position, Aspect ratio, Clipping planes and Fov) and an unsigned int “backgroundMusic” is created and then used as value for what sound to play.

* + 1. gameLoop()

While the \_gameState isn’t “EXIT” then both methods “processInput()” and “drawGame()” are called along with 3 std::outs that show if collisions occurred between each of the “mesh#” and “mesh4” using “CheckCollison()”.

* + 1. Process Input()

An SDL\_Event “evnt” is declared to hold date on the type of event. While there is an “evnt” active, a switch statement is used to determine the next step. If the “evnt” is an SDL\_KEYDOWN then add the key pressed to the std::vector using “AddKey()”, if its SDL\_KEYUP, remove it.

case SDL\_KEYDOWN: //On key down

AddKey(evnt.key.keysym.sym); //add key to the vector

break;

case SDL\_KEYUP: //on key up

RemoveKey(evnt.key.keysym.sym); // remove key from the vector

break;

Then a float “speed” is used to define the speed of movement and the key pressed determines the axis moved in.

* + 1. drawGame()

The method starts by clearing the display and then additional glm::vec3 are created: “monkeyPos”, “mokney2Pos” and “teapotPos”. The first model is then rotated to face the camera, the shader is bound, the position and scale are set and then the shaders update is called with the parameters that will be assigned to uniforms. Also the relevant texture is bound, the mesh is drawn and the corresponding bounding sphere is updated.

//Monkey - Model

shader.Bind();

transform.SetPos(monkeyPosition);

transform.SetScale(glm::vec3(1.0f, 1.0f, 1.0f));

shader.Update(transform, myCamera, glm::vec3(0.0, 0.0, 0.5), glm::vec3(1.0…

texture3.Bind(0);

mesh1.draw();

mesh1.UpdateSphereData(\*transform.GetPos(), 1.2f);

This is repeated for all meshs and then the display buffer is swapped.

* + 1. AddKey()

An iterator of the same time as the vector is created, along with a bool “false”. The vector “pressedKeys” is then looped through and if the SDL\_Keycode “key” parameter is the same as any in the vector, then “found” is set to true, if “found” is false after the vector has been looped though, add the keypress into the vector.

for (it = pressedKeys.begin(); it != pressedKeys.end(); it++)

{

if (\*it == key)

{

found = true;

}

}

if (!found)

{

pressedKeys.push\_back(key);

}

* + 1. RemoveKey()

Like the above method (1.2.8) there is an iterator of the same type as the vector. The vector “pressedKeys” is looped though but only iterated when the SDL\_Keycode “key” parameter is not the same as any item in the vector, if it is, then erase the SDL\_Keycode that the iterator points too. [GEEKSFORGEEKS]

if (\*it == key)

{

it = pressedKeys.erase(it);

}

else

{

it++;

}

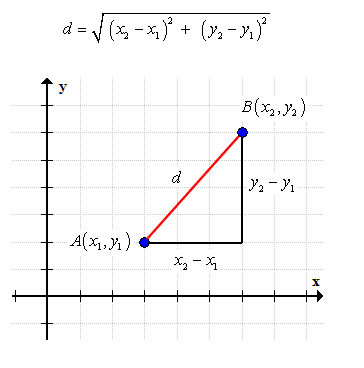
* + 1. CheckKey()

Similar to the above two methods (1.2.9 & 1.2.10)the vector is iterated through and if the “Key” passed in is found to match any in the vector then it shall return true, else false.

* + 1. CheckCollison()

This method is used to check if two bounding spheres intersect by passing in a float “radius#” and a glm::vec3 “pos#” for each respective sphere and then a distance calculation (Figure 1) is used to determine if the distance between the spheres is greater than “radius1” + “radius2”. If it is the method shall return true, if not, false.

return distance < radius1 + radius2;

  
Figure 1. Simplified Distance Calculation [CHILIMATH, 2017]

* 1. **Mesh**
     1. Variables
        1. Vertex Variables

A glm::vec3 “pos” that holds positional data.

A glm::vec3 “texCoord” that holds data regarding the texture coordinates.

A glm::vec3 “normal” that holds data on the objects normal.

* + - 1. BoundingSphere Variables

A glm::vec3 “pos” holds positional data.

A float “radius” that holds the value of the radius of the bounding sphere.

* + - 1. Mesh Variables

A GLuint “vertexArrayObject” that is used to hold the objects vertices.

A GLuint “vertexArrayBuffers” is used to hold data on the array of buffers.

An unsigned int “drawCount” holds the data on how much of the “vertexArrayObject” needs to be drawn.

A BoundingSphere “boundingSphere” to act as a spherical collider.

An enum “POSITON\_VB” used to hold the integer value of the buffer to be used.

An enum “TEXCOORD\_VB” used to hold the integer value of the buffer to be used.

An enum “NORMAL\_VB” used to hold the integer value of the buffer to be used.

An enum “INDEX\_VB” used to hold the integer value of the buffer to be used.

An enum “NUM\_BUFFERS” used to hold the integer value of the buffer to be used.

* + 1. initModel()

The “draw count” is set to the number of indices that the parameter const IndexedModel& “model” has. Then a vertex array is generated and stored in the vertextArrayObject (VAO). The next step is to bind the VAO and then generate the buffers, based off the VAO.

The next three blocks of code are similar in design, so they will be described singularly. “glBindBuffer()” is used to inform openGL on what kind of data the buffer is, then “glBufferData()” is used to send said data to the graphics processing unit (GPU). Then “glEnableVertexAttriArray()” is used to specify the index of the generic vertex attribute (GVA), this value corresponds to the location of the layout within the shader. The final step is to use “glVertexAttribPointer()” to define an array of attribute data. The first parameter in this method is used index the GVA to be modified and this is the same index used in “glEnableVertexAttriArray()”. The second is the size of components per GVA, for example the “POSITION\_VERTEXBUFFER” and the “NORMAL\_VB” both have the value of 3, representing the x, y and z respectively. Whereas the “TEXCOORD\_VB” only has an x and y, therefore the size parameter is only 2. [KHRONOSGROUP, 2018]. The next block of code is the same as the ones before however the data isn’t sent over to the shader.

glBindBuffer(GL\_ARRAY\_BUFFER, vertexArrayBuffers[POSITION\_VB]);

glBufferData( //move the data to the GPU. (Adapted structure for Documentation)

GL\_ARRAY\_BUFFER,

model.positions.size() \* sizeof(model.positions[0]),

&model.positions[0],

GL\_STATIC\_DRAW

);

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

The last step is the VAO is unbound.

* + 1. UpdateSphereData

This method is used to update the data of the boundingSphere, for example the parameters passed in are used to set the position and radius of the sphere.

* + 1. GetSpherePos()

Returns the position of the boundingSphere.

* + 1. GetSphereRadius()

Returns the radius of the bounding sphere.

* + 1. mesh()

Sets the “drawCount” to NULL.

* + 1. ~mesh()

Used to delete the VAOs.

* + 1. loadModel()

Having been passed a filename and location (as a single string), the model is set to the .obj in that file location and then it is passed to “initModel”.

* + 1. draw()

The VAO is bound and passing in the mode of “triangles”, the draw count of the model, specifying the type of values in the indices, then finally the index, which represents a pointer to the position of where the indices are stored to “glDrawElements”, draws the model. Finally, the VAO is unbound.

* 1. **Dispay**
     1. Variables

A float “screenWidth” to hold the x value of the screen.

A float “screenHeight” to hold the y value of the screen.

A SDL\_GLContext “glContext” to hold the context.

A pointer to a SDL\_Window “sdlWindow” that holds the pointer to the out window.

* + 1. Display()

The constructer sets “sdlWindow” to “nullptr”, sets the float “screenWidth” to 1024 and the float “screenHeight” to 768.

* + 1. ~Display()

The destructor deletes the glContext and destroys the sdlWindow before quitting the program with “SDL\_QUIT()”.

* + 1. returnError()

This methods takes a std::string “errorString” and then displays it in the console using std::cout before telling the user to press any key to quit and when a key is pressed, quits the program using SDL\_Quit().

* + 1. getWidth()

This method simply returns the float value of variable “screenWidth”.

* + 1. getHeight()

This method simply returns the float value of variable “screenHeight”.

* + 1. swapBuffer()

This method is used to swap the buffers of the window and therefore update the screen and achieves this using SDL\_GL\_Swapwindow.

* + 1. clearDisplay()

clearDisplay() is passed in four floats to represent the values for red, green, blue and the alpha. This values are then used in “glClearColor()” and then glClear() is used to clear the colour and depth buffers, aswell as set the colour to the colour declared in the above line.

* + 1. initDisplay()

Using SDL\_INIT\_EVERYTHING, everything is initialised. Then all the SDL\_GL\_SetAttribute methods are called, setting the minimum number of bits that can be used to display colour to 8 for red, green and blue. The SDL\_GL\_DEPTH\_SIZE is set to 16, the default number for the z-buffer and finally sets double buffering on.

Then an SDL\_Window is created by passing a name, the middle of both the x and the y values, the screen width and screen height (both cast to Ints) and the GL\_Context into “SDL\_CreateWindow()”. There are then 3 if statements used to catch errors using the method discussed in 1.4.4. Then both GL\_DEPTH\_TEST and GL\_CULL\_FACE are enabled with “glEnable()” and these enables z-buffering and means that faces that are not pointing at the camera aren’t drawn, respectively.

* 1. **Shader**
     1. Variables

A static const unsigned int “NUM\_SHADERS” that holds the number of shaders, in this case 2.

A GLuint “program” used to track the shader program.

A GLuint “shaders[NUM\_SHADERS]” to hold the array of shaders.

A GLuint “uniforms[]” to hold the number of uniform variables, in the case 8.

* + 1. init()

Firstly “program” is defined by creating a shader program that openGL will save a reference number for. Then the 2 shaders are created by loading the relevant shaders “Lighting.frag” and “Lighting.vert” from the resource folder in the project. The next step is to loop through the shaders and attach each to the program. Then the attribute variables from section 1.3.2 are associated with the shader program.

glBindAttribLocation(program, 0, "position");

glBindAttribLocation(program, 1, "texCoord");

glBindAttribLocation(program, 2, "normals");

Then “glLinkProgram()” is called to create executables that will be run on the GPU, after which “glValidateProgram()” is called to check that the program is valid. Both of these steps are checked respectively by “CheckShaderError()” that will be discussed in section 1.6.8. The final step in this method is to assign the uniforms.

uniforms[0] = glGetUniformLocation(program, "m"); //Uniform for model matrix

uniforms[2] = glGetUniformLocation(program, "lightColor"); //Uniform for light colour

uniforms[3] = glGetUniformLocation(program, "lightPos"); //Uniform for light position

uniforms[4] = glGetUniformLocation(program, "target"); //Uniform for light target

uniforms[5] = glGetUniformLocation(program, "cameraPos");//Uniform for camera position

uniforms[6] = glGetUniformLocation(program, "vp");//Uniform for view projection matrix

\*internal commentary edited to fit for document.

* + 1. ~Shader()

When the destructor is called, the shaders are looped though, first detaching the shader, then deleting it. After which the program is also deleted.

* + 1. Bind()

“bind()” uses “glUseProgram()” to declare the handle of the program object that is to be used in the rendering state.

* + 1. Update()

This method is used to update the uniform variables within the shader, therefore it takes the parameters of: the transform, the current camera, the colour of the light, the position of the light, the targets position and finally the camera position. Here is an example of updating the uniforms.

glUniformMatrix4fv(uniforms[0], 1, GLU\_FALSE, &m[0][0]); //Model matrix

glUniformMatrix4fv(uniforms[6], 1, GLU\_FALSE, &vp[0][0]); //View projection matrix

glUniform3f(uniforms[2], lightColor.x, lightColor.y, lightColor.z); //Light colour

glUniform3f(uniforms[3], lightPos.x, lightPos.y, lightPos.z); //lightPos

* + 1. CreateShader()

“glCreateShader()” is used first to create a shader based on the type passed in, in the case of this project, firstly “GL\_VERTEX\_SHADER” and then “GL\_FRAGMENT\_SHADER”. If no shader has been created, a relevant error message is sent. The next step is that the string passed in is converted to a list of c-strings, then using “glShaderSource()” and “glCompileShader” all the source code is sent to openGL and compiled. A final “CheckShaderError()” is called before the shader is returned.

* + 1. LoadShader()

A std::ifstream “file” is created to operate on the files needed, in this case the filename that was passed in. Then 2 std:strings are delacred “output” and “line”. If the file is currently open and the file is “good” (none of the streams error states are triggered), then it will extract the line and turn it into a string, before passing it into the variable “output”. “output” is then appended with a new line and returned. If the stream had any of its error states triggered, then it will inform the user with an error message.

* + 1. CheckShaderError()

When this is called, if the shaders were created without error, then the program is queried for errors. However, if the shaders encounter an error while being created, then it shall check the shaders. If the checks are unsuccessful then use “glGetProgramInfoLog()” to return the information log for the program object. Then same is done for the shader with “glGetProgramInfoLog()”. The error is then printed.

* 1. **Audio**
     1. Variables

An int “sourceID” that holds the id for the source.

An int “bufferID” that holds the id of the buffer.

A char\* that holds the buffer.

A std::string “name” that holds the name.

A float “x” that holds the x-axis positional data.

A float “y” that holds the y-axis positional data.

A float “z” that holds the z-axis positional data.

A std::vector<data> “datas” that holds the audio data.

A pointer to an ALCcontext\* “context” that holds the context.

An ALCdevice “\*device” that holds a pointer to the audio device.

* + 1. Audio()

The constructer opens the sound card, if an error is encountered then an error message is sent to the console using std::out informing the user of said error. Next a context is created with “alcCreateContext()” and once again if this is unsuccessful an error message is sent to the console. Then using “alcMakeContextCurrent” the current context is set to the created context.

* + 1. ~Audio()

When the destructor is called the std::vector “datas” is lopped through and the audio sources and their corresponding buffers are deleted. Before the current context is deleted and the audio device is closed.

* + 1. isBigEndian()

This method exists solely to check if the bytes composing the data are ordered from most significant to least, or vice versa.

* + 1. convertToInt()

If the data isn’t big endian then the data is converted to be represented by an int.

* + 1. LoadWAV()

This method is used to load the sound data from the files and returns the “soundData” upon completion.

* + 1. loadSound()

This is the method that calls “LoadWav()” along with checking if the audio is single channel or stereo. It is also responsible for generating the id for the audio and adding it to the std::vector “datas”.

* + 1. playSound()

This method uses the int identifier to find the correct audio, then it will play that audio using “alSourcePlay()”.

* + 1. playSound()b

This method functions similarly to the section above (1.6.8) however, it can also take positional data to make the sound differ depending on where in world space it is played.

* + 1. stopSound()

This methods uses “alSourceStop” and the int identifier to select the audio and then stop it from playing.

* + 1. setListener()

This is used to set the audio listener of the device and also to make the listener orientated towards a specific user specified point.

* 1. **Texture**
     1. Variables

A GLuint “textureHandler” used to hold data on the address of the texture.

* + 1. Init()

Additional variables are declared: “width”, “height” and “numComponets”. Which respectively hold the width of the texture, the height of the texture and the number of the components of the texture. Then an unsiged char\* “imageData” is created and then using “stbi\_load()” with the “fileName” passed in, it loads the image to use as a texture and stores the data [STB\_IMAGE]. There is an error check next to inform the user if the image is returned NULL. Next, we generate the texture names with “glGenTextures()”, the first parameter determines the number of names, the second where they are stored. “glBindTexture()” is then used to bind one of the previously created texture names to the desired texture, where the first parameter is the type of texture and the second is the created name. “glTexParameteri()” is called twice, once to wrap the texture outside of the width and another to do the same for the height. It is then called twice again, once to enable linear filtering for when the texture is smaller than the area it is being mapped to, referred to as minification. Then again for when the texture is larger than the target mapping area, this process is called magnification. Finally “glTexImage2D()” is used to specify a 2D texture by taking: the target texture, the mipmapping level, the pixel format and other information like: the width, height, border size, input format, the data type of texture and finally the image data.

glTexImage2D(

GL\_TEXTURE\_2D, //Target

0, //Mipmapping Level

GL\_RGBA, //Pixel Format

width, //Width

height, //Height

0, //Border Size

GL\_RGBA, //Input Format

GL\_UNSIGNED\_BYTE, //Data Type of Texture

imageData); //Image Data

\*structure of internal commentary edited for document.

* + 1. ~Texture()

The destructor uses “glDeteTextures()” which specifies how many named textures to be deleted, along with the location of said textures.

* + 1. Bind()

Firstly the macro “assert” is used to identify that the current texture is one of the 32 available textures. Then “glActiveTexture()” is used with the passed in parameter to active the specified texture unit. Lastly “glBindTexture()” is used to specify what type of texture is being dealt with and then bind the actual texture.

1. **Overview of pass through Shaders**
   1. **Lighting.vert**
      1. Layouts

A vec3 “position” to hold the position of each vertex.

A vec2 “texCoord” to hold the position of the texture coordinates.

A vec3 “normals” to hold data on the normals of the object.

* + 1. Uniforms

A mat4 “m” which holds the model matrix.

A mat4 “vp” which holds the view projection matrix.

* + 1. Variables - Out

A vec3 “Normal” to hold the new normal data.

A vec2 “texCoord0” to hold the new texture coordinates data.

A vec3 “Position” to hold the new positional data.

* + 1. Vertex Shader

The variables “texCoord0” is set equal to the “texCoord” in the layout and are passed via “out” to the fragment shader along with the new “Normal” and “Position” data. The “Normal” variable is set to the transpose, of the inverse of the model matrix, cast to a “mat3”, multiplied by the “normals”. This is to bring the normals from world space into eye space and this matrix is referred to as the “normal matrix” [Heuvel, 2011]. Then “Position” is set equal to the outcome (cast into vec3) of the model matrix multiplied by the “position”, which has been cast to a vec4. Finally, “gl\_position”, a built-in output variable, is set to the “vp” multiplied by the “Position” cast into a vec4. This was done to inform the program of the position of the current vertex in world space, and to pass over the relevant data to the fragment shader.

* 1. **Lighting.frag**
     1. Uniforms

A vec3 “lightColor” to hold the colour value of the light.

A vec3 “lightPos” to hold the position of the light.

A vec3 “cameraPos” to hold the position of the camera.

A sampler2D “diffuse” to hold the 2D texture.

* + 1. Variables – In

See sections 1.1.1 as they remain unchanged.

* + 1. Variables – Out

A vec4 “FragColour” that holds the colour of the fragment.

* + 1. Fragment Shader

The first type of lighting to be calculated is ambient, “ambientlighting”. This is done by multiplying the float “ambient Strength”, in this case 0.1, by the “lightColor”. This gives the impression of nature, indirect lighting.

The second type of lighting is diffuse, “diffuseLighting”. This is calcauted by first normalising the vec3 “Normals”. Then the target direction, “targetDirection” is worked out by using a direction calculation between the lights position, “lightPos” and the position of the vertex. The value “diff” is calculated using the “max()” of the dot product of the normalised normal and the normalised target direction. Then multiplying “diff” with the light colour, like the last lighting calculation, returns the value for diffuse lighting, “diffuseLighting”.

The “Blinn-Phong” lighting model was used over the “Phong” lighting model to enable more visually accurate specular. This is because when the angle between the view direction and the reflection direction is above 90, it nullifies the specular when using just the “Phong” model.

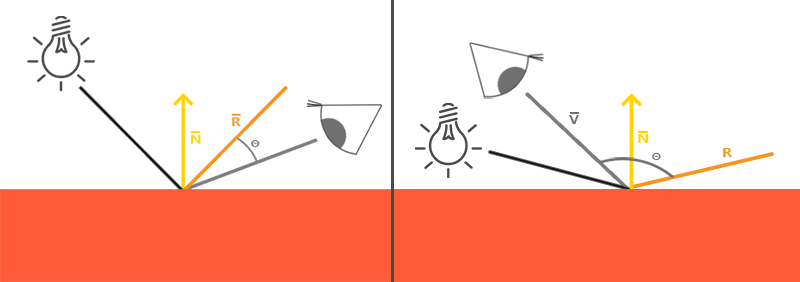
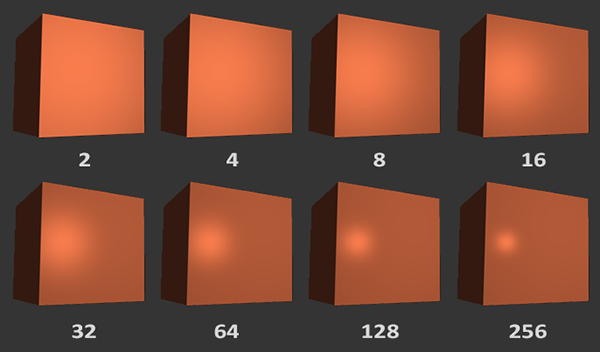


Figure 3. “Phong Lighting Model Fault” LEARN OPENGL, 2014d]

So with that said, the final type of lighting to be calculated is specular lighting, of the “Blinn-Phong” variety [LEARNOPENGL, 2014]. This is done by first setting a specular strength between 0-1, in this case, 1. Then the direction of view is calculated with another direction calculation and normalised. Next the “halfwayDir” is calculated, this is a unit vector that is halfway between the light direction and the direction of view. The specular will appear brighter and fuller the closer this vector aligns with the surface normal. The actual calculation of the specular is a “clamped dot product between the surface normal and the halfway vector to get the cosine angle between them…” [LEARN OPENGL, 2014b] the final value in this calculation is for the shininess factor.

  
Figure 2. “Visual impact of different shininess values” [LEARN OPENGL, 2014c]

This value is then multiped by the specular strength value and then the colour of the light to return the specular lighting value, “specularLighting”.

The final result is calculated by multiplying together the “ambientLighting”, “specularLighting” and “diffuseLighting” values and then casting it to vec4, and multiplying it by the “texture2D()” containing the data for the texture and the texture coordinates. This results in the colour for the fragment being returned, considering all three of the lighting methods, along with the texture of the object.

1. **References**

Based off university lab work by Bryan Young [Young, 2018].

GEEKSFORGEEKS. *Delete elements in C++ STL list – GeeksforGeeks* [online]*.* GeeksforGeeks. [viewed 3 December 2018]. Available from:

<https://www.geeksforgeeks.org/delete-elements-c-stl-list/>

CHILIMATH., 2017. *Distance Formula shown as the Derivative of the Pythagorean Theorem* [online image]. [viewed 14 December 2018]. Available from: <https://www.chilimath.com/lessons/intermediate-algebra/distance-formula/>

KHRONOSGROUP, 2018. *Khronos Combined OpenGl Registry – The Khronos Group Inc* [online]. Khronos Group. [viewed 10 December 2018]. Available from: <https://www.khronos.org/registry/OpenGL/>

CPLUSPLUS., 2017. *cplusplus.com – The C++ Resources Network* [online] cplusplus. [viewed 13 Novemember 2018]. Available from:   
<http://www.cplusplus.com/>

LEARN OPENGL., 2014a. *LearnOpenGL – Basic Lighting* [online]. Learn OpenGL. [viewed 14 November 2018]. Available from:

<https://learnopengl.com/Lighting/Basic-Lighting>

LEARN OPENGL., 2014b. *LearnOpenGL – Advanced Lighting* [online]. Learn OpenGL. [viewed 19 November 2018]. Available from:

<https://learnopengl.com/Advanced-Lighting/Advanced-Lighting>

Heuvel, S., 2012. Opengl – *What extractly mat3(a mat4 matrix) statement in glsl do? – Stack Overflow* [online] Stack Overflow. [viewed 12 December 2018]. Available from: <https://stackoverflow.com/questions/10879864/what-extractly-mat3a-mat4-matrix-statement-in-glsl-do>

LEARN OPENGL., 2014c. Visual impact of different shininess values. [online image]. [viewed 14 November 2018]. Available from:

<https://learnopengl.com/Lighting/Basic-Lighting>

LEARN OPENGL., 2014d. Phong Lighting Model Fault. [online image]. [viewed 14 November 2018]. Available from:

<https://learnopengl.com/Advanced-Lighting/Advanced-Lighting>

YOUNG, B. 2018. Games Programming 2 [lecture]. *Labs 1-7.* Glasgow Caledonian University, 30 September – 30 November.

BLY7, 2018. *Bly7/OBJ-Loader* [online]. GitHub. [viewed 7 October 2018]. Available from:

<https://github.com/Bly7/OBJ-Loader>

NOTHING, 2018. *nothings/stb: stb single-file public domain libraries for C/C++* [online]. Github. [viewed 7 October 2018]. Available from:

<https://github.com/nothings/stb>