|  |  |
| --- | --- |
| Car Game  Games Programming 2 | 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.  Scott Davidson  Matriculation Number: S1917367 |

Contents

[1. Game Audio Source Class 2](#_Toc29559885)

[2. Game Camera Struct 4](#_Toc29559886)

[3. Game Display Class 6](#_Toc29559887)

[4. LoadObject.h & .cpp 8](#_Toc29559888)

[5. Main Code of Game Class 9](#_Toc29559889)

[6. Mesh Creation Class 11](#_Toc29559890)

[7. ShaderCreationClass 13](#_Toc29559891)

[8. Stb\_image class 15](#_Toc29559892)

[9. TextureCreationClass 16](#_Toc29559893)

[10: transform.h 17](#_Toc29559894)

# Game Audio Source Class

This code inherits data and methods from the *GameAudioSource.h* header file.

Within the constructor, *GameAudioSource()*, an OpenAL device is opened. If the device variable is made null, an error message will appear to tell the user that the sound card cannot be opened. In addition to this, a context is created, and all the necessary data is stored within the context. However, as with the sound card, if the context comes back null, an error message will appear to inform the user. Finally, the specified context is made the current context within the game.

In the destructor, *~GameAudoSource(),* the OpenAL data that was created earlier is destroyed. The alDeleteSources deletes all audio sources. The alDeleteBuffers function deletes all the buffers and then all associated data is also deleted. Afterwards, the context is deleted, and the audio device is closed.

The raw data is handled by two methods, *isBigEndianUsed()* and *convertToIntegerType()*. This is essential as it is needed to load the sounds.

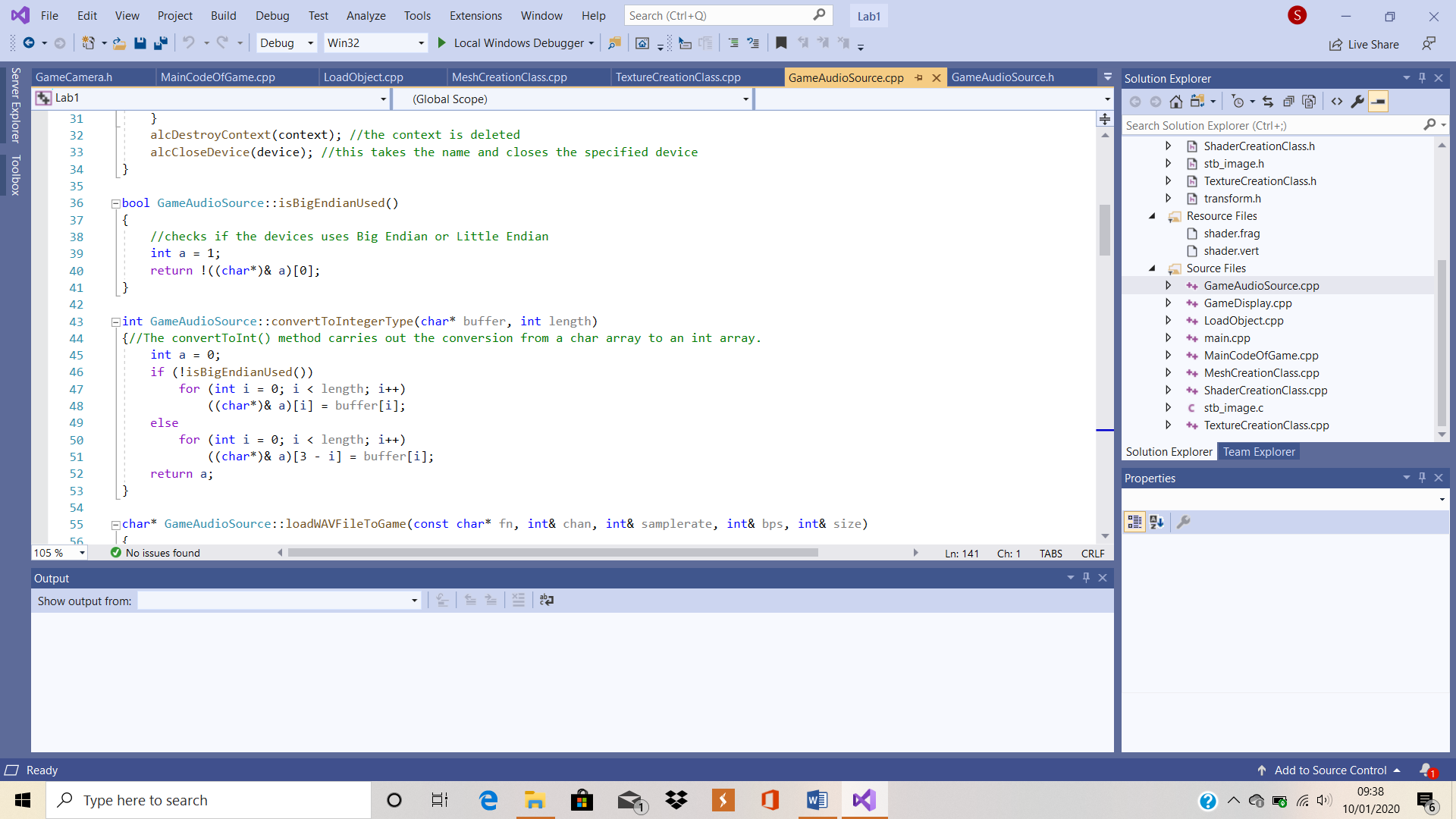


Figure 1: isBigEndianUsed & convertToInt

Two different ways that a computer stores data is Big Endian and Little Endian. When Big Endian is used, the byte with the highest level of significance is given the lowest address and the rest of the data is placed in order and contained in the next three bytes in memory. In contradiction to this, when Little Endian is used, the byte with lowest level of priority is given the lowest address and the rest of the data is placed in order and contained in the next three bytes in memory. This is important as, to ensure the sound data is read properly, it needs to be clear which the current device uses. As show in Figure 1, the *isBigEndianUsed()* method checks which the current device uses.

WAV files are stored as char (character) type data. However, OpenAL only uses variables of type integer as parameters. As shown in Figure 1, the *convertToIntegerType()* method carries out the conversion from a char array to an int array.

The *loadWAVFileToGame (const char\* fn, int& chan, int& samplerate, int& bps, int& size)* method interprets the WAV file so it is possible to play it back using openAL.

The *loadSoundToGame(const char\* filename)* follows a pathway to find a file and loads it into the game. The file is then loaded in and it takes note of important pieces of data, such as sample rate and the type of audio, then saving it in a buffer that can be used at any time. Each piece of data is then given an integer value to identify them and these integers are then used to reference each audio track, and therefore an unsigned int is used as a return type.

The *playGameSound (unsigned int id, glm::vec3& pos)* method uses the integer handler to decide which audio track to play and the predefined 3D position to decide where to play the sound from. It uses this information to play the appropriate sound at the right time and from the right place.

The *stopGameSound(unsigned int id)* method uses the id of the audio tracks to decide which track to stop at what time.

Finally, the *setAudioListener(glm::vec3& pos, glm::vec3& camLookAt)* method determines the position of the audio listener and sets what part of the game the listener is looking at. The position of the listener is the same as the position of the camera. If the camera were to be moved during gameplay, the *camLookAt* vector would become equal to the camera forward vector.

An attempt to swap the background music audio file from the one used in the labs was made but was unsuccessful as, when the game was run, only static was heard despite the file being loaded properly.

# Game Camera Struct

The code for this struct is based on work we completed in the labs but has been changed to fit the context of the game. For example, the left, back, MoveUp and MoveDown methods have been added and called in another class to allow the user to move the camera by pressing certain buttons. There is no code involved in the constructor method of this struct, *GameCamera().*

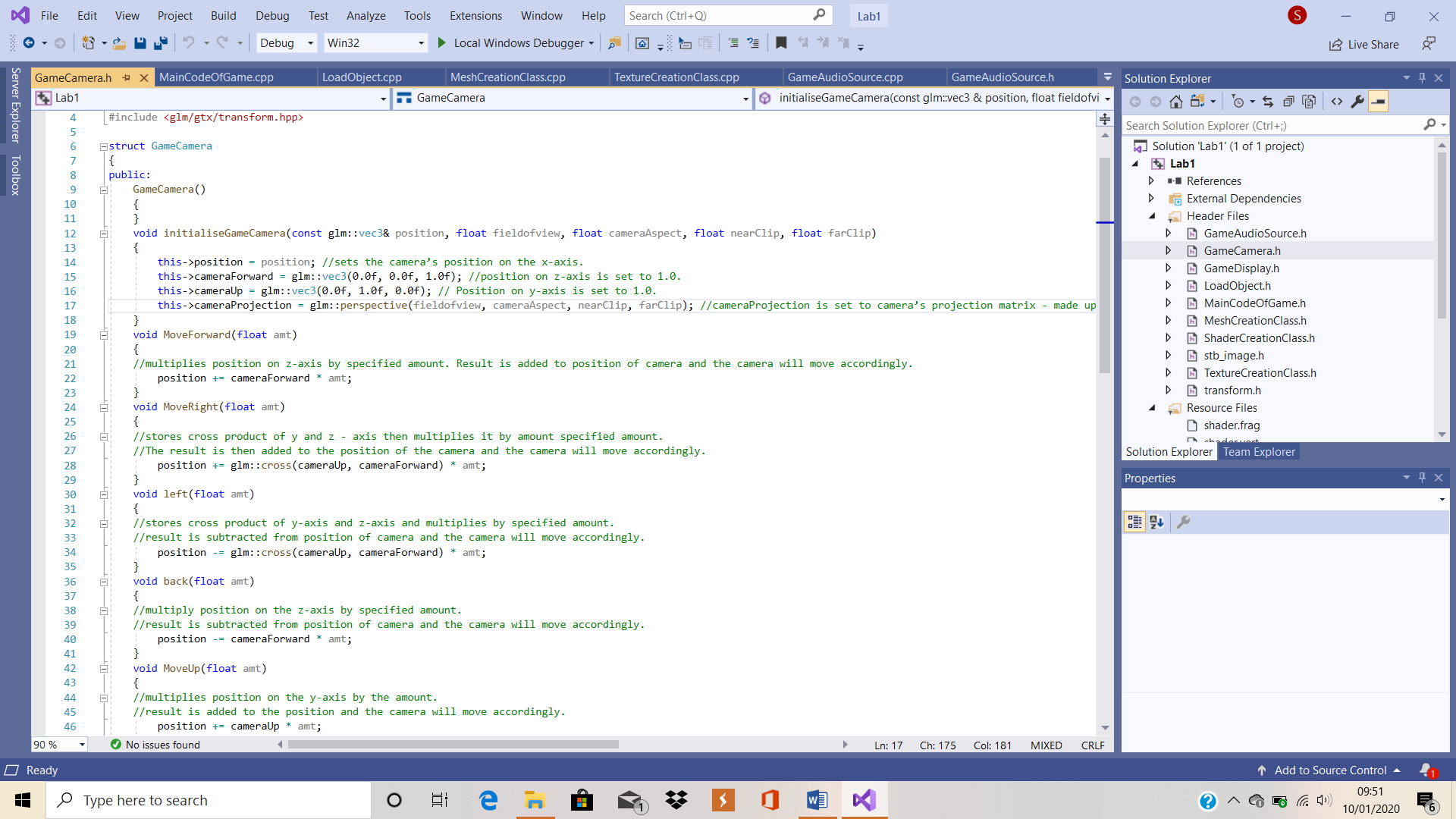


Figure 2: GameCamera struct

Figure 2 shows examples of the code implemented in to this header file. The first method with code, void *initialiseGameCamera(const glm::vec3& position, float fieldofview, float cameraAspect, float nearClip, float farClip)* method starts by setting the camera’s position on the x-axis. The position on the z-axis, also known as the cameraForward variable, is then set to 1.0 on the z-axis. The position on the y-axis, also known as the cameraUp variable, is set to 1.0 on the y-axis. The cameraProjection variable is set to the camera’s projection matrix, which is made up of the camera’s field of view, the aspect ratio, the near clip and the far clip.

The void *MoveForward (float amt)* method works by multiplying the position on the z-axis by the specified amount. The result is then added to the position of the camera and the camera will move accordingly.

The void *MoveRight(float amt)* method works by storing the cross product of the y-axis and the z-axis and then multiplying it by the amount specified amount. The result is then added to the position of the camera and the camera will move accordingly.

The newly implemented *void left(float amt)* method works by storing the cross product of the y-axis and the z-axis and then multiplying it by the specified amount. The result is then subtracted from the position of the camera and the camera will move accordingly.

The newly implemented *void back(float amt)* method works by multiplying the position on the z-axis by the specified amount. The result is then subtracted from the position of the camera and the camera will move accordingly.

The newly implemented *void MoveUp(float amt)* method works by multiplying the position on the y-axis by the specified amount. The result is then added to the position of the camera and the camera will move accordingly.

The newly implemented *void MoveDown(float amt)* method works by multiplying the position on the y-axis by the specified amount. The result is then subtracted from the position of the camera and the camera will move accordingly.

The glm::vec3 *getPosition()* method returns the camera’s position to the game.

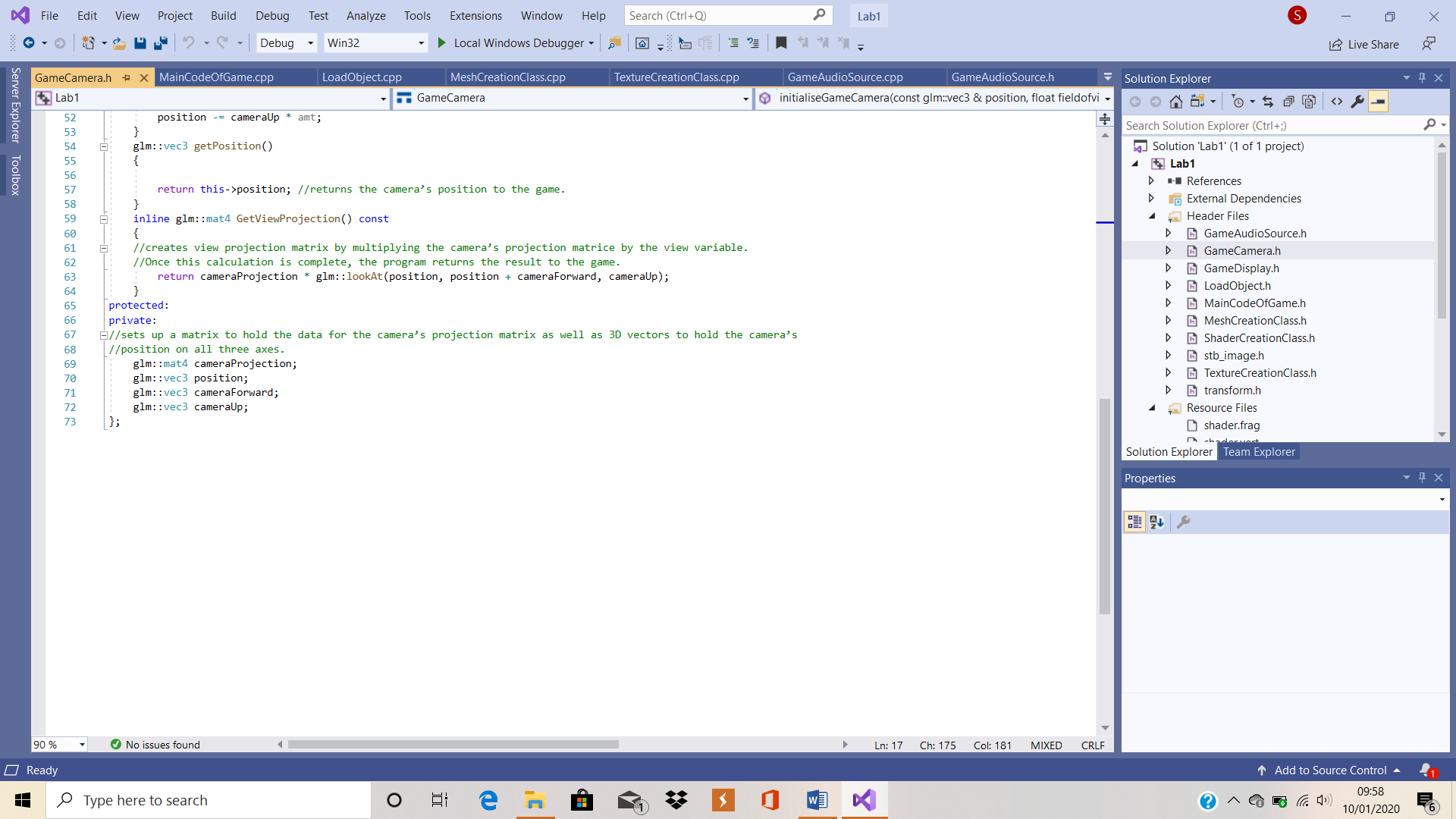


Figure 3: Bottom Of Header File

As shown in Figure 3, the *inline glm::mat4 GetViewProjection() const* method creates a view projection matrix by multiplying the camera’s projection matrice by the vec3 view variable. Once this calculation is complete, the program returns the result to the game.

As also shown in Figure 3, at the bottom of the header file, the program sets up a matrix to hold the data for the camera’s projection matrix as well as 3D vectors to hold the camera’s position on all three axes.

# Game Display Class

This code inherits data and methods from the *GameDisplay.h* header file. This class was obtained during the implementation of the lab work and the variables have been changed.

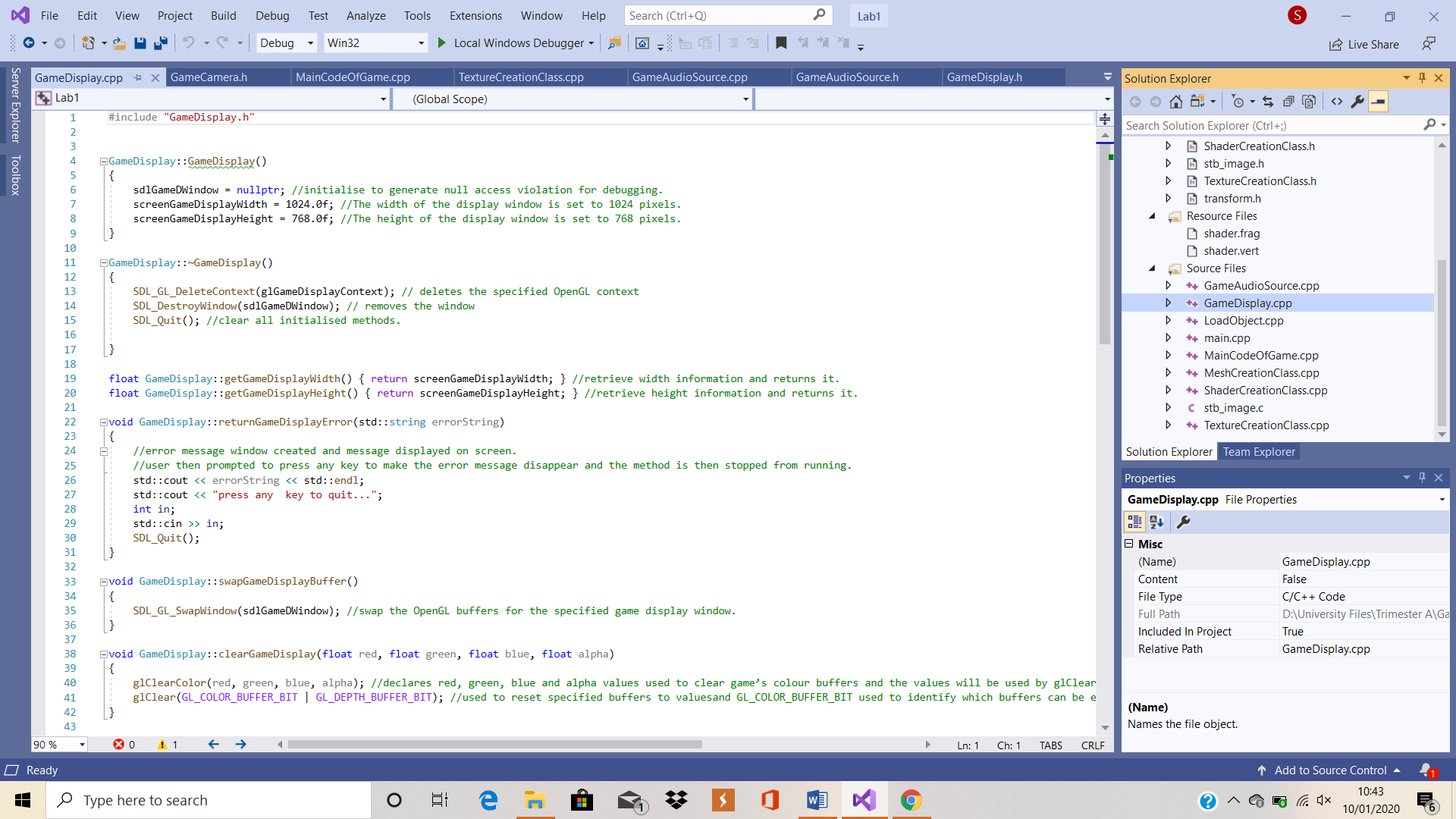


Figure 4: Start of .cpp file

Figure 4 shows contents of the .cpp file, which includes all the code from the constructor to the *ClearGameDisplay()* method.

Within the constructor, *GameDisplay()*, the game’s display window is assigned a nullptr. In C++, the word nullptr is used to display a null pointer value and a null pointer value is used to declare that no handles or pointer types point to an object, in this case, the game display window. The width of the game display window is then set to 1024 pixels and the height is set to 768 pixels.

In the destructor, *~GameDisplay()*, the display context is deleted using the *SDL\_GL\_DeleteContext* function. Next, the game display window is then destroyed using the *SDL\_DestroyWindow* function and, finally, the *SDL\_Quit* function is used to clear all initialised methods.

The float *getGameDisplayWidth()* and *getGameDisplayHeight()* methods are getter methods that retrieve the height and width information and return them to the game.

In the *returnGameDisplayError()* method, an error message window is created and a message detailing any errors is displayed on screen. The user is then prompted to press any key to make the error message disappear and the method is then stopped from running.

In the *swapGameDisplayBuffers()* method, the *SDL\_GL\_SwapWindow* function is used to swap the OpenGL buffers for the specified game display window.

In the *ClearGameDisplay(float red, float green, float blue, float alpha)* method, glClearColor declares the red, green, blue and alpha values that will be used to clear the game’s colour buffers and these values will be used by glClear. The glClear function is used to reset the specified buffers to the predetermined values and, in this case, GL\_COLOR\_BUFFER\_BIT is used to identify which buffers can be edited and GL\_DEPTH\_BUFFER\_BIT indicates the depth buffer.

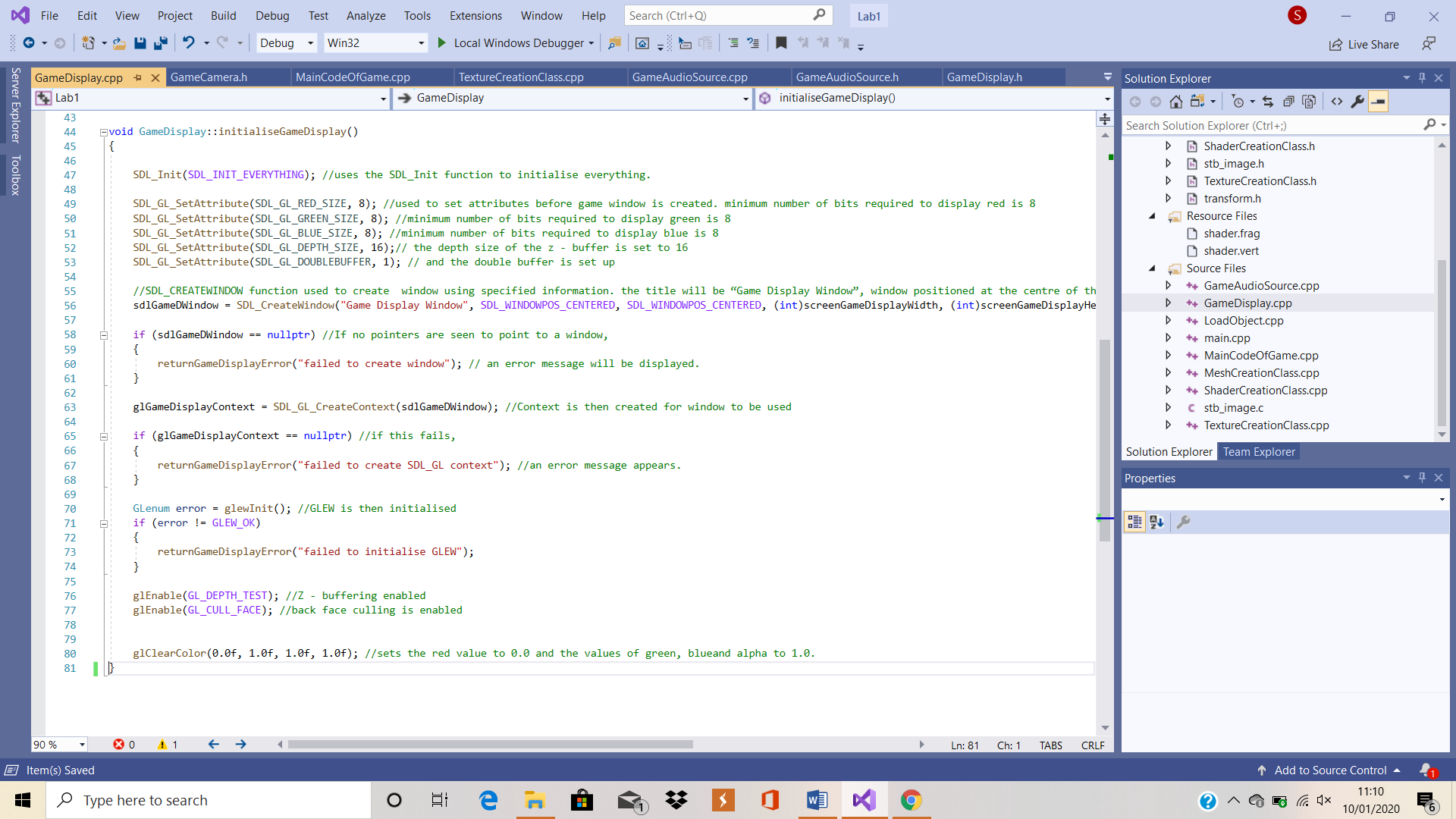


Figure 5: InitialiseGameDisplay()

Finally, as Figure 5 clearly shows, the *initialiseGameDisplay()* method uses the SDL\_Init function to initialise everything. After this, the SDL\_GL\_SetAttribute is used to set attributes before the game window is created. In this case, the minimum number of bits used to display red, blue and green are all set to 8, the depth size of the z-buffer is set to 16 and the double buffer is set up. Next, the SDL\_CREATEWINDOW function is used to create a window using the specified information. In this case, the title of the window shall be “Game Display Window”, the window will be positioned at the centre of the screen and it will the dimension that were specified earlier. If no pointers are seen to point to a window, an error message will be displayed.

A context is then created for the window to be used and, if this fails, an error message appears. GLEW is then initialised. Z-buffering and back face culling are then enabled. Back face culling makes sure that faces that the camera is not pointing at are not drawn. Finally, the glClearColor sets the red value to 0.0 and the values of green, blue and alpha to 1.0.

# LoadObject.h & .cpp

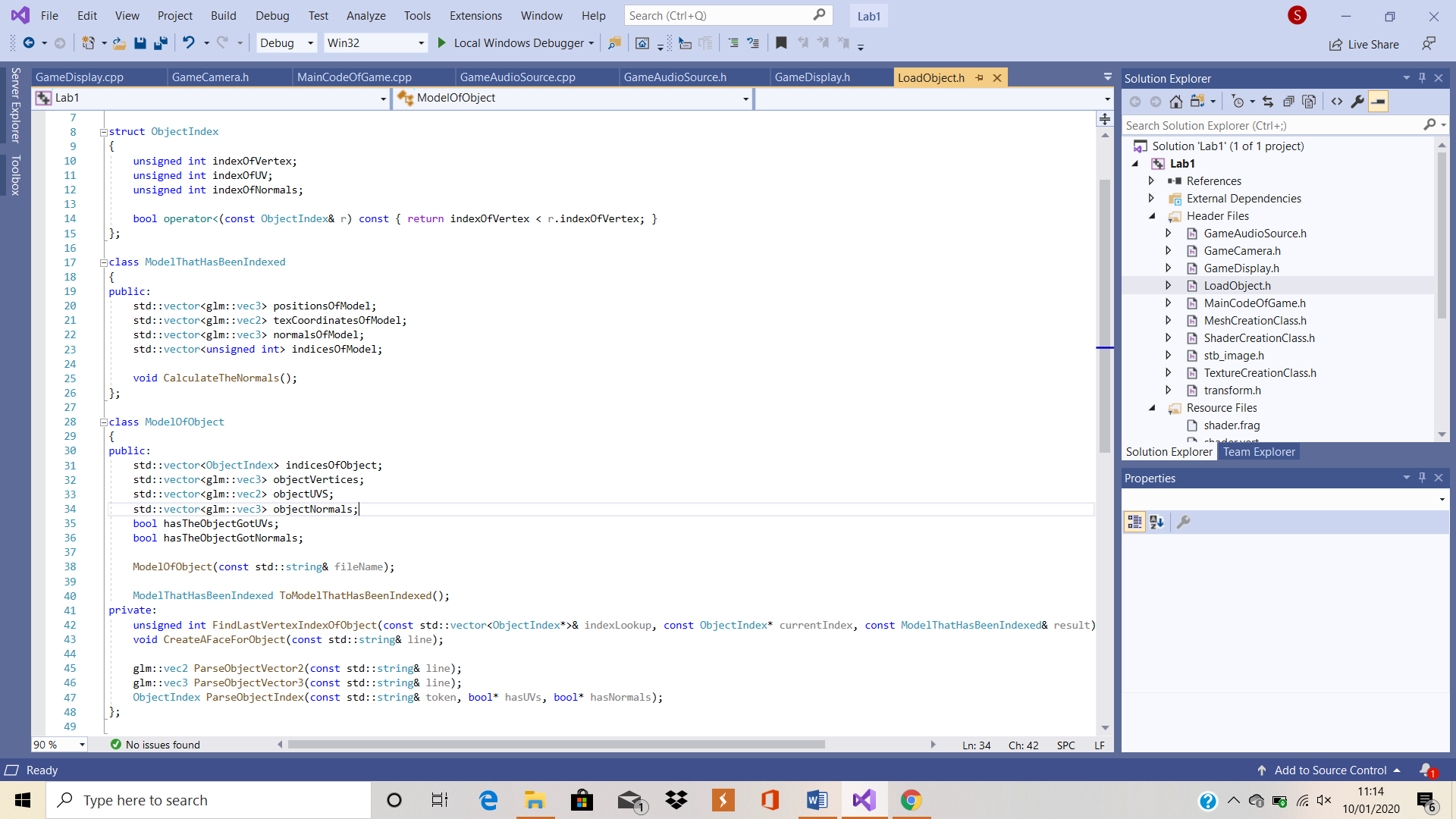


Figure 6: LoadObject.h

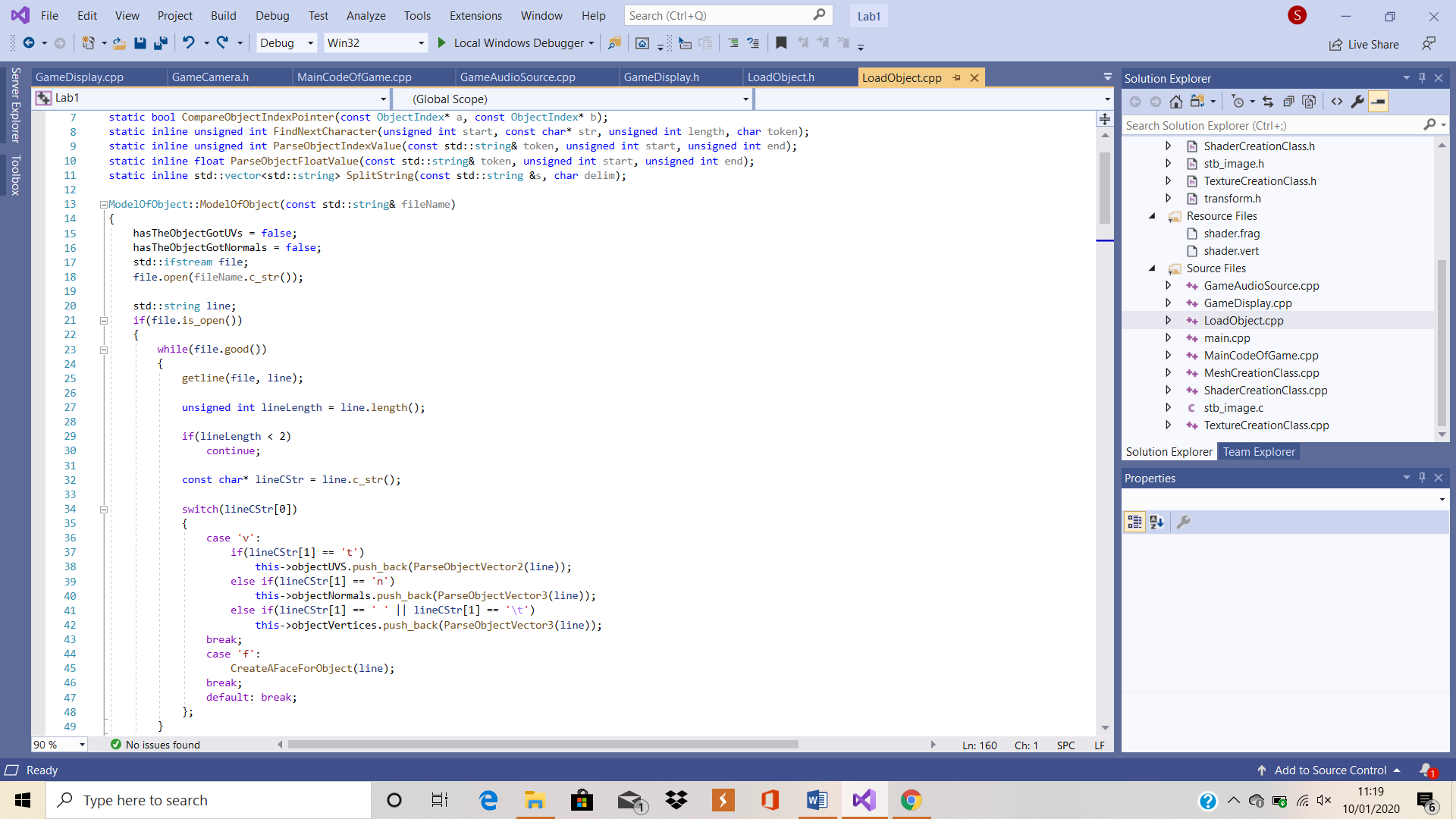


Figure 7: LoadObject.cpp

Figure 6 shows the contents of the methods within the header file and Figure 7 shows the beginning of the .cpp file. This class was created during the implementation of the lab work and remains unchanged in structure as it is thought to be the best fit for the solution. However, the variables and classes have been changed accordingly.

# Main Code of Game Class

This cpp files inherits methods and variables from both the *MainCodeOfGame.h* & *GameCamera.h* header files. This class was obtained during the implementation of the lab work and many changes have been made to the code and layout.

In the constructor, *MainCodeOfGame()*, the state of the gameplay is changed from exit to playing. Secondly, a new game display is created using the GameDisplay() class information. Thirdly, three new mesh variables are created calling information from the MeshCreationClass class. The three variables are called carMesh, houseMesh & tireMesh. As the names suggest, the variables will be used for the car, house and tire models. Finally, an audio device is created using information in the GameAudioSource class and will be used to deal with the audio later.

Unlike most classes in this game, there is no code contained in the destructor of this class, *~MainCodeOfGame()*. In the *runMainCodeOfGame()* method, the code calls and runs the *initMainGameSystems()* method. In addition to this, the *MainGamePlayLoop()* method is also called and run.

In the *initMainGameSystems()* method, firstly, the *initialiseGameDisplay()* method is called from the GameDisplay class and runs it to initialise the display. Next, the soundEffect integer variable uses the audio device to load the bang sound effect to the game from its file. Similarly, the backgroundMusic integer variable uses the audio device to load the background music to the game from its file. Next, the three mesh variables run the *loadModelToGame()* method to load models in from file by following the pathway specified. In this case, the files are being loaded from the res folder and, as expected, the car mesh will load in a car model, the house mesh will load in a house model and a tire model will be loaded in by the tire mesh variable. Finally, the myGamePlayCamera variable calls the initialiseGameCamera method from the GameCamera struct and uses it to set up the camera with the specified position, field of view, camera aspect, near clip and far clip.

In the *MainGamePlayLoop()* method, while the gameplay is still running, the *processMainGameInput()* method is called and run, along with the *drawMainGameToScreen()* method. The *collisionDection()* method is also called and run. However, changes are made to this method. The mesh1Position and mesh1Radius are made to equal the position and radius of the car mesh, the mesh2Position and mesh2Radius are made to equal the position and radius of the house mesh and the mesh3Position and mesh3Radius are made to equal the position and radius of the tire mesh. Finally, the *playAudioInGame(backGroundMusic, glm::vec3(1.0f,5.0f,6.0f))* method is run, with the source of the audio becoming the background music file and the position of the source becoming 1.0 on the x-axis, 5.0 on the y-axis and 6.0 on the z-axis.

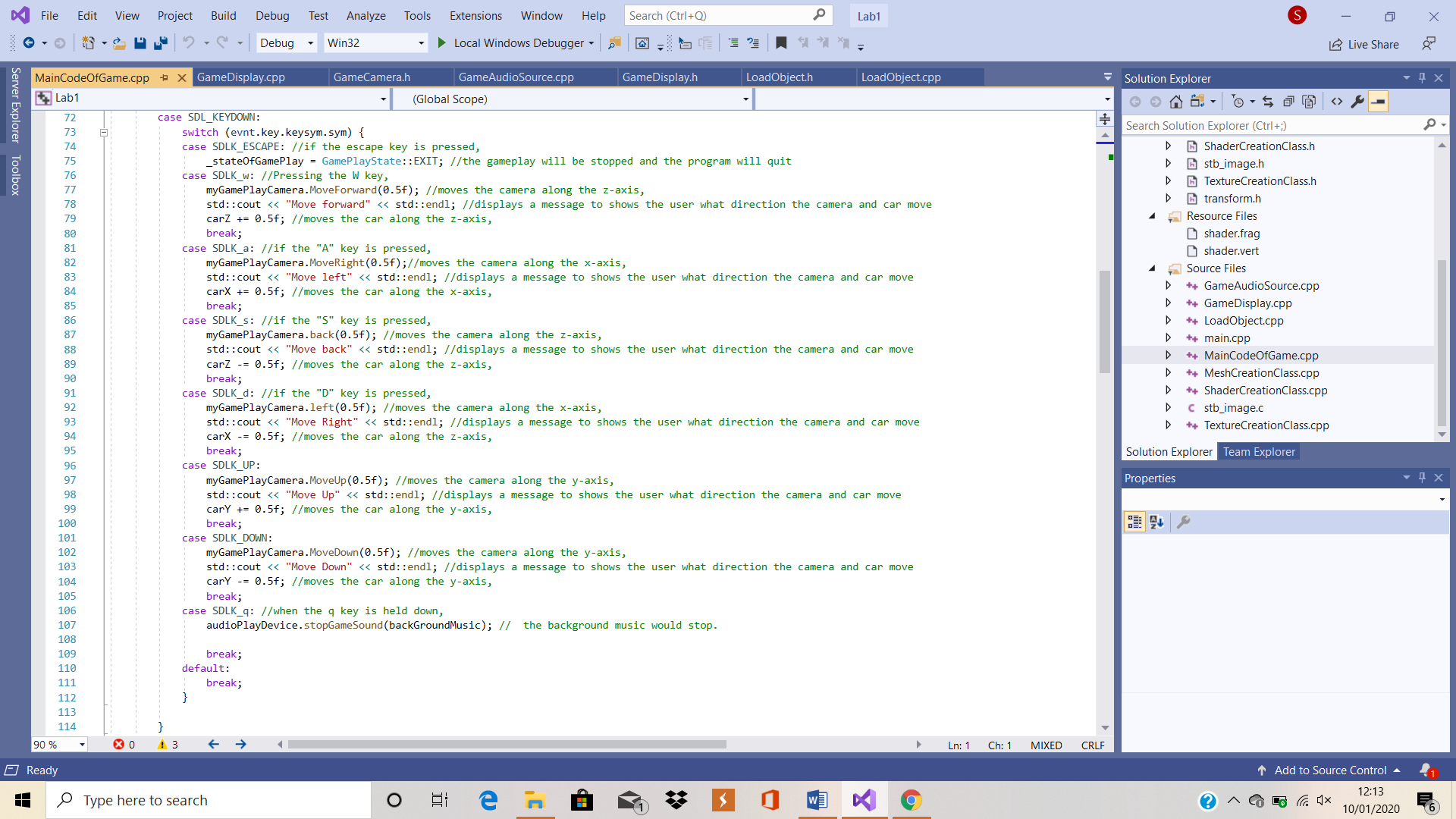


Figure 8: W, A, S, D controls

In the *processMainGameInput()* method, all key presses and inputs are handled. If the red X at the top of the screen is pressed, the state of gameplay will be changed to quit, and the game will stop. The program is then set up to carry out certain tasks depending on if specified buttons are pressed down. As shown in Figure 8, this game follows the control system of many popular PC games with the main control buttons being W, A, S and D. Pressing the W key moves the camera and car along the z-axis, pressing the A key moves the camera and car positively along the X-axis, pressing the S key moves the camera and car along the z-axis, and, finally, pressing the D key will move the camera negatively along the X-axis. Pressing the UP arrow moves the camera and car up the screen on the y-axis and pressing the DOWN arrow does the inverse. The camera and car move simultaneously as to give the illusion to the player that the camera is fixed on the car or attached in some fashion. Code was added so that, when the q key is held down, the background music would stop.

In the *collisionDetection()* method, the distances between all three of the models are calculated. The distances are calculated using the distance = the square root of the total sum of (x2 – x1) squared plus (y2 – y1) squared plus (z2 – z1) squared. This calculation is run, calculating the distance between the meshes. If the first and second meshes collide, the sound effect will play from the position of the first mesh, if the first and third meshes collide, the sound effect will play from the position of the first mesh and, If the second and third meshes collide, the sound effect will play from the position of the second mesh.

In the *playAudioInGame(unsigned int Source, glm::vec3 pos)* method, when the method is called, the state will change to AL\_PLAYING and the game will play the audio source file, either the soundEffect or backgroundMusic variables, from the specified position within the game.

Finally, in the *drawMainGameToScreen()* method, the *ClearGameDisplay()* method is called and the values of the variables are altered to make the background yellow. Next, a shader is created, and the red texture is loaded in to the game. The position, scale and rotation of the first model are then set. The position of the car is set to equal the values of 3 float variables declared in the header file for this class. The shader and texture are then attached to the model, which is then drawn in to the game. Finally, once the model is drawn, the *updateSphereData()* method is called to continuously detect collisions. This procedure is repeated for all models, but the textures will vary depending on the model. The process input method is called and run to enable the models to move after being created. The colour array is then enabled for writing. Finally, the *swapGameDisplayBuffers()* method is used to swap the OpenGL buffers for the specified game display window.

# Mesh Creation Class

This cpp files inherits methods and variables from the *MeshCreationClass.h* header file. This class was obtained during the implementation of the lab work and the variables have been changed.

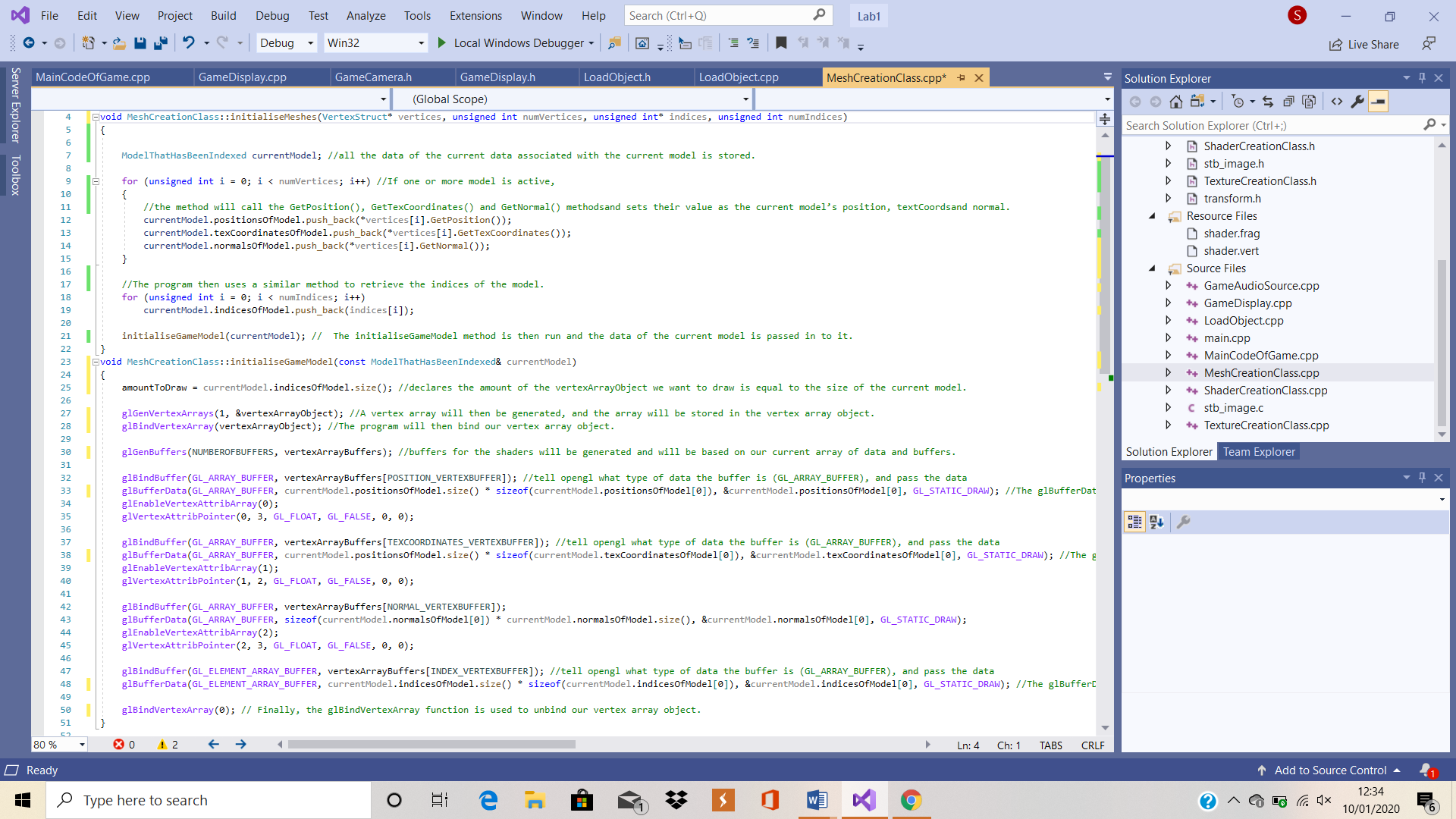


Figure : initialiseMeshes & initialiseGameModel

Figure 9 shows the setup of the first two methods within the MeshCreationClass.cpp file. At the start of the *initialiseMeshes(VertexStruct\* vertices, unsigned int numVertices, unsigned int\* indices, unsigned int numIndices)* method, all the data of the current data associated with the current model is stored. If one or more model is active, the method will call the GetPosition(), GetTexCoordinates() and GetNormal() methods and sets their value as the current model’s position, textCoords and normal. The program then uses a similar method to retrieve the indices of the model. The initialiseGameModel method is then run and the data of the current model is passed in to it.

The *initialiseGameModel(const IndexedModel& currentModel*) method starts by declaring that the amount of the vertexArrayObject we want to draw is equal to the size of the current model. A vertex array will then be generated, and the array will be stored in the vertex array object. The program will then bind our vertex array object. The buffers for the shaders will be generated and will be based on our current array of data and buffers. The glBufferData moves data to the GPU. The information contained in the brackets represents the type of data being passed and its size, the data’s pointer and where the data is stored on the GPU. Finally, the glBindVertexArray function is used to unbind our vertex array object.

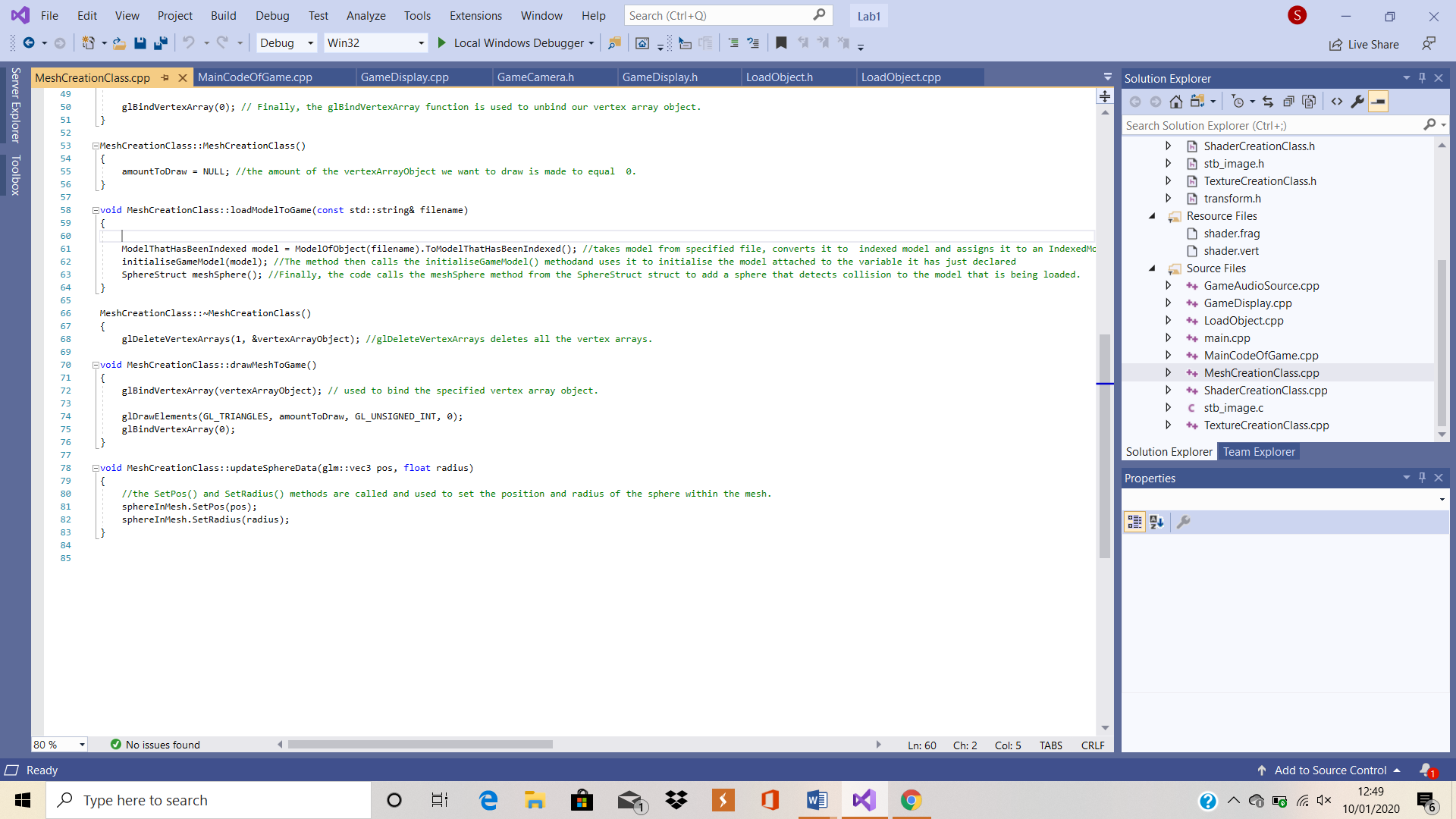


Figure : Bottom of MeshCreationClass.cpp

Figure 10 shows the layout of the code in the following methods. In the constructor for this class, *MeshCreationClass(),* the amount of the vertexArrayObject we want to draw is made to equal 0.

The *loadModelToGame(const std::string& fileame)* method takes a model from a specified file and converts it to an indexed model and assigns it to an IndexedModel variable called model. The method then calls the initialiseGameModel() method and uses it to initialise the model attached to the variable it has just declared. Finally, the code calls the meshSphere method from the SphereStruct struct to add a sphere that detects collision to the model that is being loaded.

In the destructor of this code, *~MeshCreationClass(),* the glDeleteVertexArrays function deletes all the vertex arrays. However, for this to work, the number of vertex array objects to be deleted is specified along with the address of the array in which the objects are contained.

In the *drawMeshToGame()* method, glBindVertexArray is used to bind the specified vertex array object. In this case, the vertex array object is called vertexArrayObject.

Within the void *updateSphereData(glm::vec3 pos, float radius)* method, the SetPos() and SetRadius() methods are called and used to set the position and radius of the sphere within the mesh.

# ShaderCreationClass

This cpp files inherits methods and variables from the *ShaderCreationClass.h* header file. This class was obtained during the implementation of the lab work and the variables have been changed

The ShaderCreationClass(const std::string& filename) method begins by creating a new shader program. OpenGL will save this as a reference. The shadersArray array is then used to create vertex and fragment shaders for the game. When one or more shaders are created, they will all be added to the shader array. The glBindAttribLocation function is used to associate our position, texCoordinate and normal variables with our shader program. The shader program is then linked which, in turn, will create executables that will be run on the GPU shaders. If this is not the case, an error message is displayed onscreen. The glValidateProgram checks to see if the shader program can run based on the current OpenGL state. If not, an error message is displayed onscreen describing the cause of the error. Finally, the glGetUniformLocation function is used to obtain and display the location of the transform uniform variable.

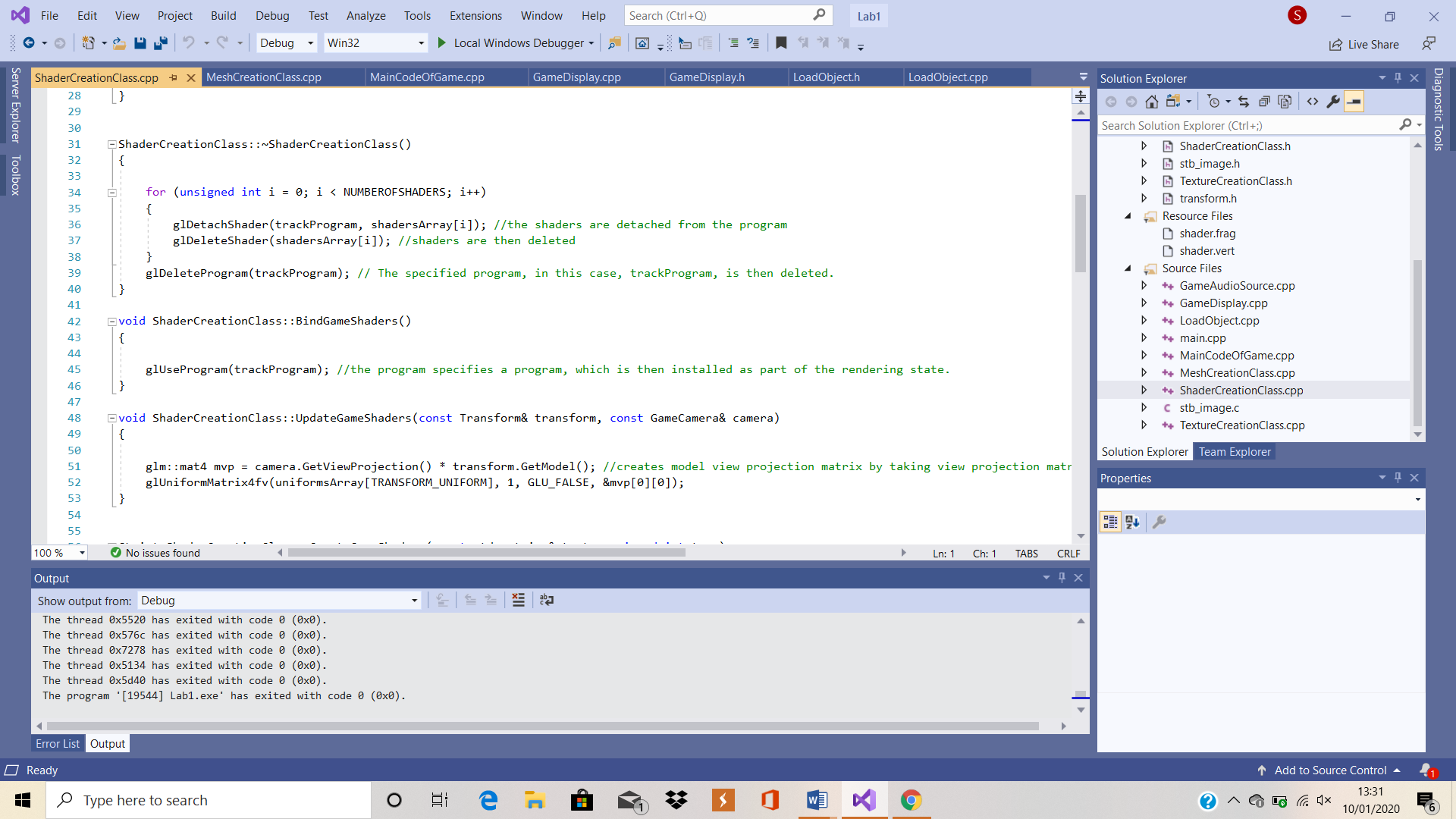


Figure : Sample code from .cpp

Figure 11 shows the code implemented in to the methods in the .cpp file for the class. In the destructor, ~ShaderCreationClass(), the shaders are detached from the program and then deleted. The specified program, in this case, trackProgram, is then deleted.

Within the void BindGameShaders() method, the program specifies a program, which is then installed as part of the rendering state.

The UpdateGameShaders(const Transform& transform, const GameCamera& camera) method creates a model view projection matrix by taking the view projection matrix it receives from the camera and multiplying it by the model data it receives from the transform header files. The command glUniformMatrix4fv is used to change an array of matrices, in this case, the array of transform uniforms. The use of 4 shows that it’s a 4x4 matrix and the command uses a series of variables and types. The first of these variables is the location, in this case, uniformsArray{transform\_uniform], which is stored as a GL\_int. The next variable taken in to consideration stores the data on the count of objects stored in the array, in this case, 1, which is stored as a GLsizei. The third variable stores the data regarding the transpose, stored as a GL Boolean and, finally, the fourth variable is a constant GL\_float called value.

The CreateGameShaders(const std::string& text, unsigned int type) method starts by creating a shader of the type that is specified by the program. However, if it fails to do so, an error message will be displayed. Any strings associated with the shaders will be converted in to lists of c-strings. The source code then gets sent to OpenGL and OpenGL will then compile the shader code. If it fails to accomplish this task, an error message will be displayed. The shader is then returned to the game.

The LoadGameShaders(const std::string& fileName) method loads in and opens a texture from file and loads it as a string. When the file is open and is functioning appropriately, the program will read every line and output it to the game, taking a new line where appropriate. If the shader fails to load, the game will display an error message. The output is then returned to the game.

The CheckForShaderError(GLuint shader, GLuint flag, bool isProgram, const std::string& errorMessage) method checks the shader program to see if the creation has been successful. If not, it sends back an error message detailing the error and sets the success variable to false.

# Stb\_image class

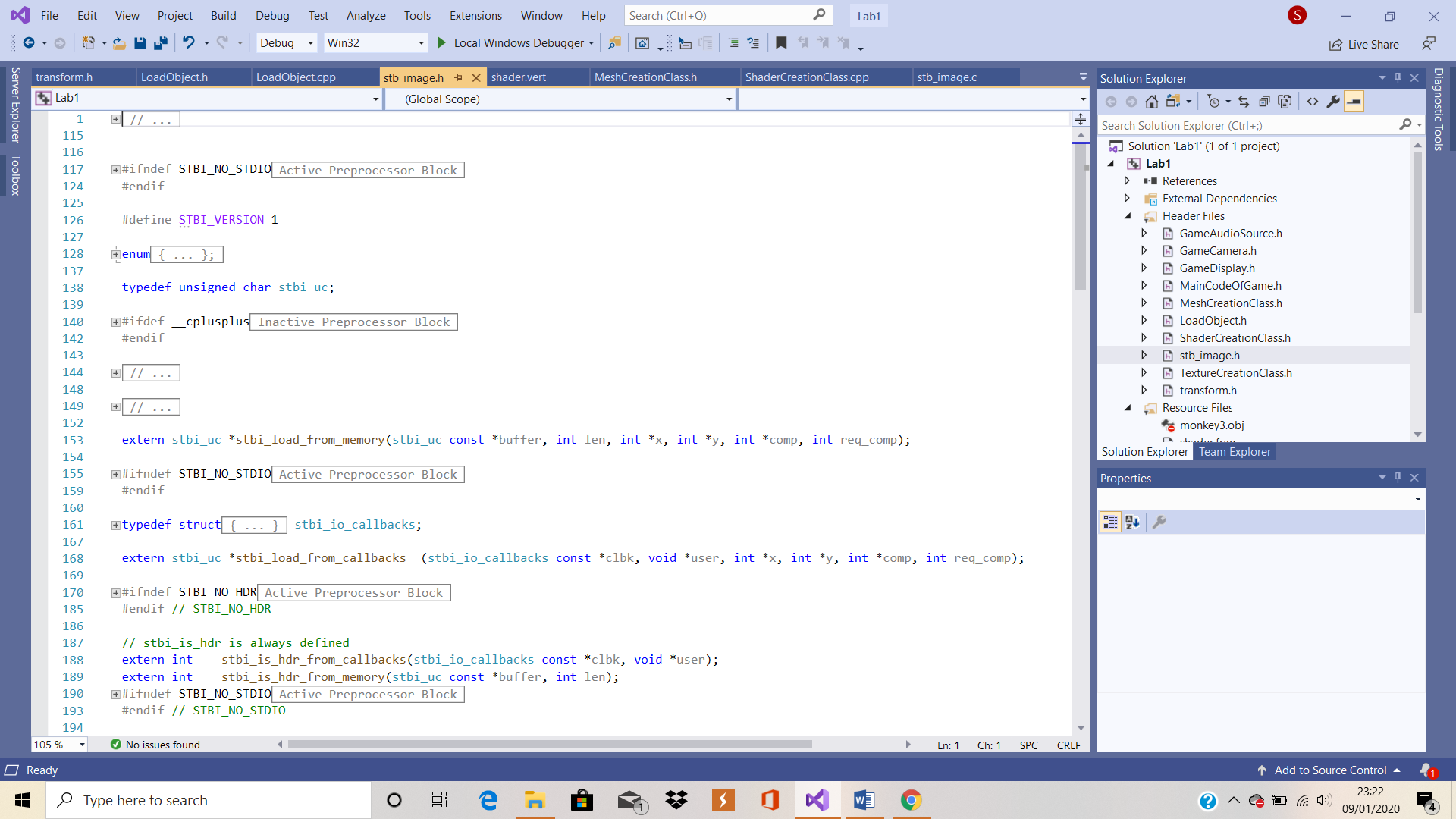


Figure : stb\_image.c

Figure 12 shows the beginning of the stb\_image.c file. This header file was obtained during the implementation of the lab work and remains unchanged as I have no knowledge of how to code using the C programming language.

# TextureCreationClass

This file inherits from the TextureCreationClass and stb\_image header files. This class was obtained during the implementation of the lab work and the variables have been changed.

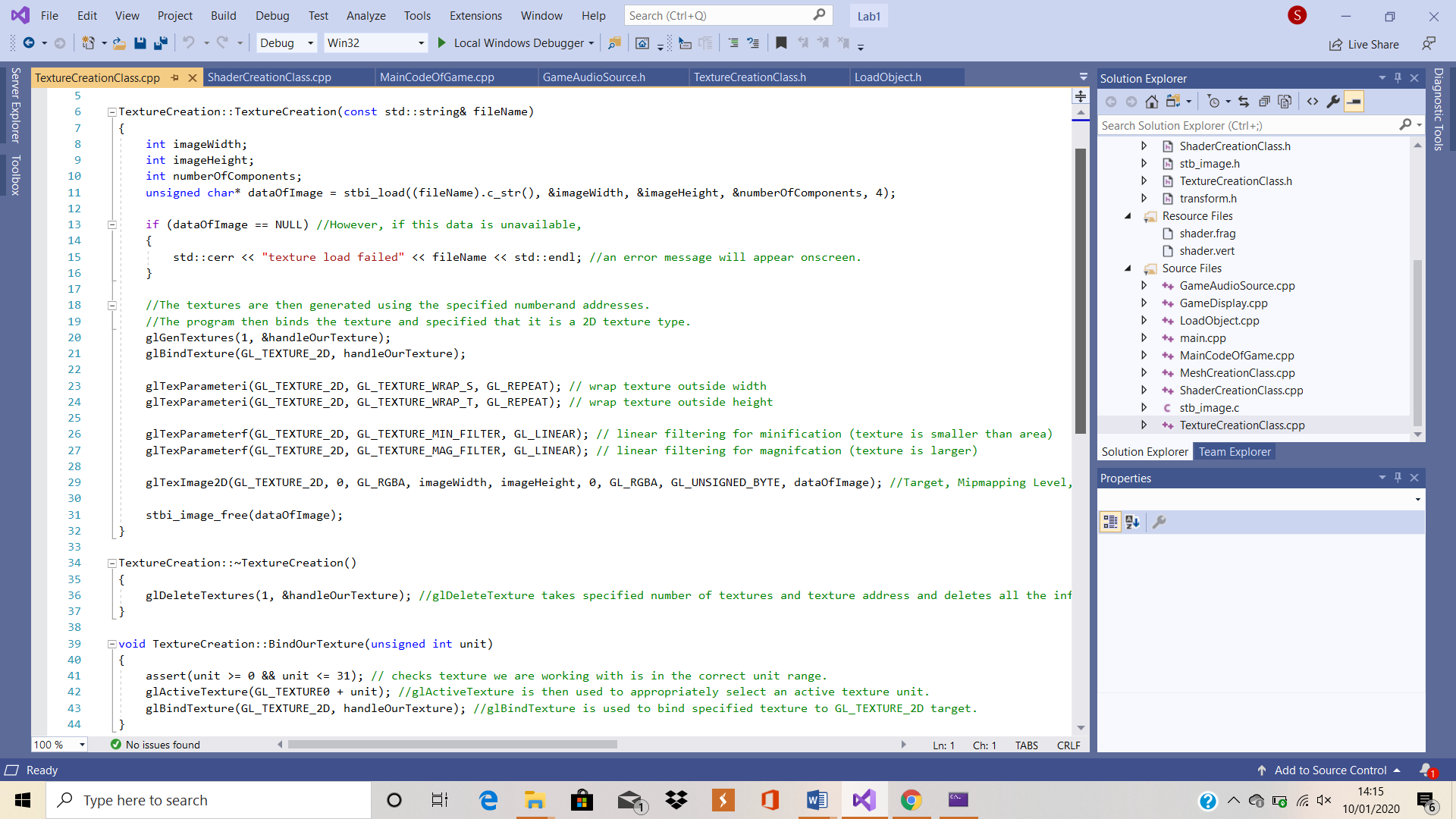


Figure 13: TextureCreationClass.cpp

Figure 13 shows the structure and code involved in the implementation of this class.

In the class constructor, TextureCreation (const std:: string& fileName), the image is loaded and data, such as the height, width and number of components, are stored. However, if this data is unavailable, an error message will appear onscreen. The textures are then generated using the specified number and addresses. The program then binds the texture and specified that it is a 2D texture type.

In the class destructor, ~TextureCreation(), the glDeleteTexture function takes the specified number of textures and the texture address and deletes all the information stored at each address.

# 10: transform.h

This header file was obtained during the implementation of the lab work and remains unchanged as it is thought to be the best fit for the solution. This code inherits data and methods from the *GameCamera.h* header file.

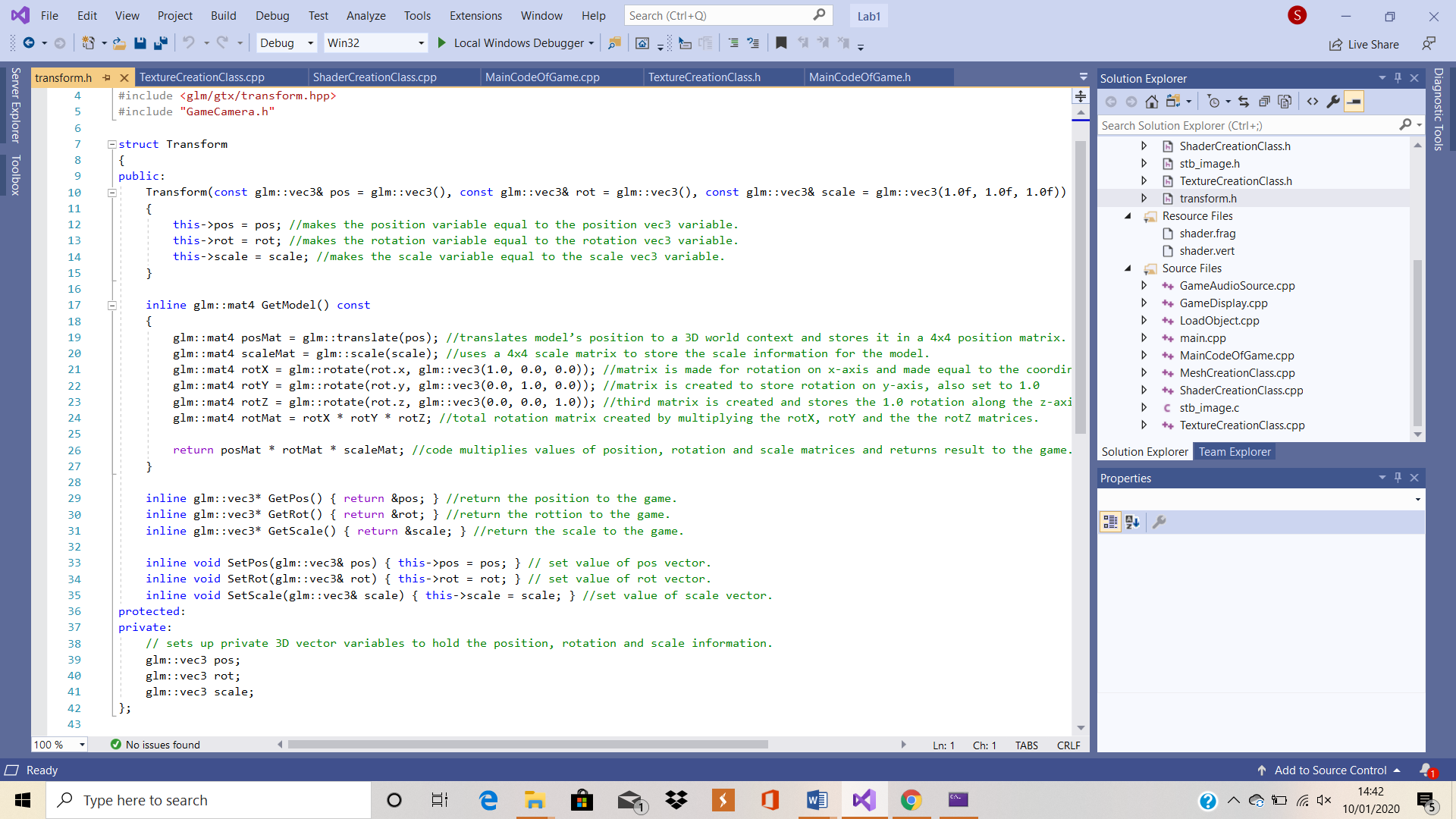


Figure 14: Transform Struct

Figure 14 shows the entire code implemented in to the Transform struct.

The constructor for this class, *Transform(const glm::vec3& pos = glm::vec3(), const glm::vec3& rot = glm::vec3(), const glm::vec3& scale = glm::vec3(1.0f, 1.0f, 1.0f))*, starts off by making the position variable equal to the position vec3 variable. The rest of the constructor makes the rotation variable equal to the rotation vec3 variable and makes the scale variable equal to the scale vec3 variable.

The *inline glm::mat4 GetModel() const* method starts by translating the model’s position in to a 3D world context and stores it in a 4x4 position matrix. The code then uses a 4x4 scale matrix to store the scale information for the model. A matrix is made for the rotation on the x-axis and is made equal to the coordinate 1.0 on the x-axis. A second matrix is then created to store the rotation along the y-axis, which is also set to 1.0 and, finally, a third matrix is created and stores the 1.0 rotation along the z-axis. A total rotation matrix is then created by multiplying the rotX, rotY and the the rotZ matrices. Finally, the code then multiplies the values of the position, rotation and scale matrices and returns the result to the game.

The *inline glm::vec3\* GetPos()* method is used to return the position to the game.

The *inline glm::vec3\* GetRot()* method is used to return the rotation to the game.

The *inline glm::vec3\* GetScale()* method is used to return the scale to the game.

The *inline void SetPos(glm::vec3& pos)* method is used to set the value of the position vector.

The *inline void SetRot(glm::vec3& rot)* method is used to set the value of the rotation vector.

The inline void SetScale(glm::vec3& scale) method is used to set the value of the scale vector.

At the bottom of the header file, the program sets up private 3D vector variables to hold the position, rotation and scale information.