Megathon-2018

BoxRunner2D

Saksham Mittal
Om Sitapara
Harshit Patel
Adithya Hosapate

September 30, 2018

Contents

1 Game Engine **4**
- 1.1 ArrowsIoEngine class 4
- 1.2 Camera2D Class 4
- 1.3 Errors class 4
- 1.4 GLSLProgram class 4
- 1.5 ImageLoader class 5
- 1.6 InputManager class 5
- 1.7 IOManager class 5
- 1.8 picoPNG class 5
- 1.9 ResourceManager class 5
- 1.10 sprite class 5
- 1.11 SpriteBatch class 6
- 1.12 TextureCache class 6
- 1.13 Timing class 6

2 Game (ArrowsIo) **6**
- 2.1 main.cpp 6
  - 2.1.1 processString() 7
- 2.2 MainGameServer class 7
  - 2.2.1 MainGameServer::run() 8
  - 2.2.2 MainGameServer::initSystems() 8
  - 2.2.3 MainGameServer::gameLoop() 8
  - 2.2.4 MainGameServer::updateChars() 9
  - 2.2.5 MainGameServer::updateBullets() 9
  - 2.2.6 MainGameServer::updateHearts() 10
  - 2.2.7 MainGameServer::updateNoPlayer() 10
  - 2.2.8 MainGameServer::processInput() 10
  - 2.2.9 MainGameServer::drawGame() 11
- 2.3 MainGame class 11
- 2.4 Bullet class 11
  - 2.4.1 Bullet::Draw() 11
  - 2.4.2 Bullet::update() 11
  - 2.4.3 Bullet::getDamage() 12
- 2.5 Character class 12
  - 2.5.1 Character::setData() 12
  - 2.5.2 Character::damageTaken() 12
  - 2.5.3 Character::increaseHealth() 12
  - 2.5.4 Character::getData() 12
  - 2.5.5 Character::draw(SpriteBatch spriteBatch) 13
  - 2.5.6 Character Movement functions 13
- 2.6 Heart class 13
  - 2.6.1 hearts::draw(ArrowsIoEngine::SpriteBatch spriteBatch) . 13
  - 2.6.2 hearts::updateTimer() 14
  - 2.6.3 hearts::setVisiblity(bool v) 14

2.7 Level class 14
    2.7.1 Level::draw() 14
2.8 Sockets class 14
    2.8.1 socketClient::receiveBytes(char *buffer) 15
    2.8.2 socketClient::sendBytes(char *buffer) 15
    2.8.3 socketServer::sendData(std::string s) 15
    2.8.4 socketServer::receiveData(std::string s) 15
    2.8.5 socketServer::select_activity() 15
2.9 Player header file (Player.h) 16

1 Game Engine

We have made our own game engine for creating window, rendering images / textures, translating the camera, managing input from user, calculating FPS (frames per second), error handling, etc.

Using GLEW for rendering graphics :
The OpenGL Extension Wrangler Library (GLEW) is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

Using SDL library for input management :
Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL.

A brief description of different classes in the game engine:

1.1 ArrowsIoEngine class

We create a namespace of ArrowsIoEngine and include all the functions of game engine in this namespace. This class contains init() function, which initialises SDL and set it's attributes.

1.2 Camera2D Class

This class manages the camera on the player. This updates the camera position according to the player movement (Camera translation and Camera scale).

1.3 Errors class

This is responsible for showing errors in case we enchanter any error. fatalError() function is called with the error string as parameter, and the error is printed on the terminal.

1.4 GLSLProgram class

It compiles the shader files vertex shader, and fragment shader and links them to the main program. It also defines use() and unuse() functions to enable and disable the Vertex attribute array.

1.5 ImageLoader class

This class is responsible for loading png's to buffer and later decoding them. It returns a GLTexture.

1.6 InputManager class

This class manages the input given by the user, it also distinguishes between when a key is pressed once and when a key is held down. For this, we are using a key map which checks whether the key map of given keyID is true and wasKeyDown() is false, then the key is pressed only once. This allows us to shoot only one bullet, even if the user keeps the mouse button pressed continuously. The update() function loops through the key map, and copy it over to the previous key map which is required for the wasKeyDown() function.

1.7 IOManager class

This class is required to read the file to a buffer. The buffer is a vector of unsigned char. It is resized to the fileSize and the file is read using read() function.

1.8 picoPNG class

The picoPNG function, decodes a PNG file buffer in memory, into a raw pixel buffer. It can convert a PNG to raw pixel data either converted to 32-bit RGBA colour or with no colour conversion at all.

1.9 ResourceManager class

This class is responsible for getting the texture from the texture path. This calls the getTexture() function from the TextureCache class.

1.10 sprite class

This class renders the sprites. each sprite can be seen as composed of triangles, so the vertexData array is set according to the coordinates and dimensions of the sprite. After this the sprite is bind to the buffer and drawn using glDrawArrays() function.

1.11 SpriteBatch class

The textures are stored in Glyphs, which is a struct containing the details of the texture. This allows us to render the same texture sprites together (called minimum texture switching). Before that, the glyphs are sorted based on texture.

1.12 TextureCache class

This class returns GLTexture, and inserts a new texture only if there is no same texture in the map. Otherwise it uses the cached texture (hence makes rendering of same textures fast).

1.13 Timing class

The main function of this class is to calculate fps and set the fps. The fps is calculated by dividing the frames in time intervals and then taking the average.

2 Game (ArrowsIo)

2.1 main.cpp

Takes input of name, character choice, (client or server) choice from user. This data is concatenated in a string to be send to all client and server. A thread named sockThread is created.

If the user chooses to become a client (i.e. choice = 2), the server's IP address is asked. The client calls the constructor "socketClient client(ip, SOCK_PORT, 2048)" to attempt connecting to the server.

If the client is connected to the server, then the server sends a welcome message (for which the client waits to receive through receiveBytes() function.

After connection is established the client sends it's info (name, character choice, server or client).

Similar to the current client, the strings from all clients and server is concatenated and sent to everyone (clients and server).

This concatenated string received by the client contains info. of all the clients and server, and is used to render other players on the client screen. This process of extracting information of other players from the received string is done using processString() function.

Then we create an instance of mainGame constructor of class mainGame and call mainGame.run().

On the other hand if the user chooses to become server (choice = 1), then number of clients is asked. Then, the server attempts to create a server with specified port number, number of clients and connection type (Non blocking in our case).

Then, same as client, it sends it's info. in the form of a string in a concatenated form.

It then creates a thread, which waits for all the clients to connect to the server, and the main thread is waiting on a while loop (while(server.init)). When all clients are connected, the sockThread changes init to false, which takes the main thread out of the busy wait.

The server also then receives the data (info. of all other players concatenated in a string) and calls processString() function.

Then we create an instance of mainGame constructor of class mainGameServer and call mainGame.run().

[NOTE: mainGame class is meant for handling clients and mainGameServer is meant for handling the server. This is because of the difference in behaviour of the two.]

2.1.1 processString()

As the name suggests, this function processes the strings received by server and clients and extracts the data present in the string. The data is separated by '|', and a simple while loop is used to extract the info. from the string. This extracted info. contains the latest data of other players (position, bullets, health, etc.), and is pushed to the players vector to be drawn on the screen.

2.2 MainGameServer class

In the constructor of MainGameServer, the initialisation of parameters takes place. The player dimensions and bullet dimensions are initialised in the constructor.

2.2.1 MainGameServer::run()

This function is responsible for running the game. It first calls initSystems() which initialises the game with all the parameters.

Then, the info. of the player is sent to all the other users, and '0|' is concatenated to the string, which shows that initially 0 bullets are fired.

gameLoop() is called after that which manages the rest of the logic of the game.

2.2.2 MainGameServer::initSystems()

This function first initialises our game engine'ArrowsIoEngine', and then creates a window of the given screen width and screen height. After that the shaders are initialised, which basically include compiling our shader files ('colorShading.vert', and 'colorShading.frag'), adding attribute for position, vertexColor, and uv coordinates.

After this, the _hearts vector is filled with the position coordinates obtained after reading the hearts.txt file.

After this the level is initialised, having a m_currentLevel variable allows to have multiple level files and rendering them just by changing the value of this variable. After this the fps counter is initialised and the character info. is pushed to the m_chars vector.

We keep a separate m_mainPlayer variable to keep track of the details of the main player. This allows us to access the parameters of our main player.

2.2.3 MainGameServer::gameLoop()

This is the main game logic which updates the characters info. and manages the data received and sent to other players. The while loop runs till it's game state is not EXIT. First we check that the player is still alive or not, and if not the processInput() is not called and the window is closed using SDL_Quit().

After this the camera position is set to the m_mainPlayer, and update() function is called (which updates the camera position as the player moves). Then the following functions are called in series:

updateNoPlayer()
updateChars()
updateBullets()
updateHearts()

[NOTE: These are defined after this function. They are responsible for updating the info. of the player and the level.]

After this a call to drawGame() is made. This is responsible for rendering the game from the info. available. After rendering the game, the updated info. of the player is sent, as the player could have moved or shooter some bullets. because of this reason, a variable called 'newBullCount' is concatenated to the string which is the number of bullets fired. newBulls is a string which contains the info. of each bullet in the format (x_pos y_pos | x_dim y_dim |). This allows other clients to get the updated info. of the server.

If the frameCounter reaches 10000, the fps is printed on the screen, and frameCounter is again initialised to 0.

2.2.4 MainGameServer::updateChars()

This function is used to parse the string which is received by the server to extract the info. of all other players. For this, we store the string 'data' (which is received in receiver() function) to tempData. We use locks to prevent race condition as the receiver function may be updating the data many times, and the value of tempData may be inconsistent due to this.

After this, we use loops to extract the info. (We have used '|' as separators between different infos. like 150.00 130.00 | 25.00 30.00 which are x_pos y_pos | x_dim y_dim respectively.)

While checking whether bullets are fired or not, we check whether the number extracted after health is 0 or not. If it is 0 then no bullets were fired, and there is no need to further traverse the string as there will be no data for bullets. But in case it is a non-0 number, we have to traverse the string further to extract the position of bullets and direction. This data is pushed back to the _bullets vector.

After all the data is extracted, we set the data for m_chars vector using setData() function. But we don't do this for the m_currentIndex (index of the main player) as the main player will have the most updated data with itself and only need the data for other players.

2.2.5 MainGameServer::updateBullets()

This function serves two main roles : checking whether a bullet has collided with a player (and if yes, the health of the player is reduced) and checking whether a bullet has hit a wall (and if yes, that bullet is popped back). This is done for all players and hence the inner loop for checking the bullet collisions runs for each player.

2.2.6 MainGameServer::updateHearts()

This function serves the purpose of updating the hearts visibility and increasing the player's health incase it is low. For each heart, it checks whether the given main player is on the same position as the heart, and if it is so it increases the health and changes the visibility. After that we run a loop to check the visibility of all the hearts, and if it is false, we call updateTimer() function.

[NOTE: See hearts class for more details on intermediate function calls.]

2.2.7 MainGameServer::updateNoPlayer()

This function keeps on updating the number of live players, and uses the function call to getLife() which return whether the player is live or not.

2.2.8 MainGameServer::processInput()

We are using SDL library for input management (either through keyboard or from mouse), support for other input devices like joystick can also be added in the future.

This function consists of a while loop which binds the type of event with corresponding physical action like mouse click, mouse movement, key press, key up, etc.

Then we check if the key pressed is Left mouse button, because that click is responsible for shooting bullets. If that is the case, we take the use coordinates using getMouseCoords() function and subtract them from the position vector of the player to get the direction vector.
After that we check which gun type the player has selected, and render the sprite according to that.

For the Arrow gun type, we calculate the angle of the direction vector using atan() function and load the corresponding png (from the 360 pngs for each degree of shooting).

After this we check similar checks whether 'w', 'a', 's', or 'd' is pressed and call the corresponding moveUP(), moveDOWN(), moveLEFT(), or moveDOWN() function). Pressing 'q' scales the screen, or in other words allows us to zoom in. The opposite happens on pressing ‘e’.

2.2.9 MainGameServer::drawGame()

This function is responsible for rendering the game on the server’s screen with the info. available to it. It sets the camera matrix and calls the draw() function of levels class to render the level.

[NOTE: See Level class for more details on rendering the level from a text file.]

Before begging to render graphics, we call _spriteBatch.begin() function. First we render the characters (only if they are alive, which we check using getLife() function). After that we render the bullets which are stored in the _bullets vector. Then, the hearts are rendered based on their visibility.

After this we call _spriteBatch.end() function after we are finished rendering.

2.3 MainGame class

The function calls are same as that of the server except for the fact, that it manages the clients. Also it uses receiveBytes() and sendBytes() function call to share data to the server.

2.4 Bullet class

The constructor of bullet class assigns the position, lifetime, direction, speed, bullet type, and texture id of the bullet. Also based on the type of the bullet, the dimensions are set accordingly.

2.4.1 Bullet::Draw()

This function class the sprite batch.draw() and subtracts the dimensions from the position while sending the parameters for positions at which it should draw.

2.4.2 Bullet::update()

First it changes the position vector by adding the speed of bullet per frame multiplied by direction vector to itself. Then, it decrements the lifetime of the bullet by 1. And if the lifetime comes 0 it returns because it will not be rendered and there is no point in checking for collisions.

However, if lifetime is not 0, it checks for collisions by checking whether the character in level data is '.' or not, because while rendering our level text file, we consider the '.' as air. We do the same checking in all directions.

2.4.3 Bullet::getDamage()

This function returns the 1st index _damage array (which is 2) if the bullet type is 3, otherwise it returns _damage[0] (which is 1).

2.5 Character class

This class is responsible for managing the character's position, health, movement and rendering the characters.

2.5.1 Character::setData()

As the name suggests, this function sets the x and y coordinate of the player and also it's health.

2.5.2 Character::damageTaken()

This function reduces the health of the player by the quantity 'damage'. If the health becomes less than 0, we update the life variable to false (meaning the player is dead). After this the rank of the player is printed on it's terminal.

2.5.3 Character::increaseHealth()

This function increases the health of the player when it is called. The health is increased only if it is less than 7 (which is the maximum health possible).

2.5.4 Character::getData()

This function returns the info. of the player (which is a character, hence this function is in character class) in the form of a string. the format of the string is as follows:

x_pos y_pos | health | index_of_heart |

2.5.5 Character::draw(SpriteBatch spriteBatch)

This function draws the character when a spriteBatch is supplied to it. It uses draw() function of spriteBatch class.

2.5.6 Character Movement functions

Character::moveUP(), Character::moveDOWN(), Character::moveLEFT(), Character::moveRIGHT() are responsible for player movement.

These functions check whether there is any other character other than '.' in the direction of motion. if that is the case, then it means that there is a wall or obstacle in the direction of motion of character.

For this, we calculate a minimum distance, and if the difference of tile width and position of character is less than this minimum wall distance, then the function returns without updating any position. This is how collision is taken care of!

If there is no such obstacle then the position is updated according to to the speed of the character in the direction of motion.

2.6 Heart class

The class manages the rendering of the hearts at different positions, and changing their visibility once a player takes a heart. The constructor takes an index as a parameter for specifying which index of heart we want to work on (changing visibility and setting timer).

We read from the text file 'hearts.txt' which contains the position of hearts (the hearts are represented by ‘‘ in the text file). We read line by line from the file and each string of line is pushed in a vector of string named m_levelData. Then, we run a loop for each character, and whenever we find '@‘ symbol we push the coordinates in the heartPosition vector. This allows us to render the hearts later.

2.6.1 hearts::draw(ArrowsIoEngine::SpriteBatch spriteBatch)

This function draws a single heart at the _position.x and _position.y which we get using the heartPosition vector at 'index' index.

2.6.2 hearts::updateTimer()

This function updates the _timer variable (incrementing it when the hearts visibility is set to false). So, our hearts wait for certain frames till they are rendered again. Once it reaches the required time, it is set to 0 (i.e. _timer = 0) and visible set to true.

2.6.3 hearts::setVisiblity(bool v)

This function sets the visibility of the heart to v (which is a bool variable), when it is called.

2.7 Level class

This class manages the rendering of levels of the game, which it reads from a text file. The text file can be customised to render any type of level (with any arrangement of bricks and grass). In the constructor of the Level class we take path of file, screen width, and screen height as input. Then, we assign the texture id's to the different textures we are going to use for rendering the level (At this point of time, we are using two types of bricks, grass and floor for drawing the level). The file is rad line by line and each line is stored in a string, and these strings is stored in a vector of strings named m_levelData.

Then we traverse this vector of strings and check for different characters. If we get a 'R', brick2 is rendered, 'L' renders brick 1, 'G' renders grass texture, and '.' renders floor texture.

2.7.1 Level::draw()

This function renders the level by calling renderBatch() function on m_spriteBatch.

2.8 Sockets class

This is responsible for managing the connections between server and clients, and transferring of data between them. The main class is Socket, and socketClient and socketServer are derived from them (socketClient is for clients and socketServer is for server).

2.8.1 socketClient::receiveBytes(char *buffer)

This function makes the client wait till it receives the data from server, and stores that data in res (of long data type). It uses the function recv() from WINSOCK library.

2.8.2 socketClient::sendBytes(char *buffer)

This function attempts to send the data to the server using send() function from WINSOCK library. And if the send fails, a fatal error is produced.

2.8.3 socketServer::sendData(std::string s)

This function is responsible for sending data to clients by the server. This is protected using mutex locks to prevent race condition.

2.8.4 socketServer::receiveData(std::string s)

This function is called to get the data from other clients and is stored in a string named 's'.

2.8.5 socketServer::select_activity()

This function maintains the complete connection between the clients and server until the game is running. Here first of all we have a loop running while the connected clients is greater than 0. First we clear the fdset for the clients to read. Also, we set the attribute readfds in FD_ZERO so that it initialises the file descriptor and set fdset to have zero bits for all file descriptors. Then we add the master scores to the fdset. Then we have a while loop which adds the clients to the fdset.

Then we wait for activity on any of the sockets, and keep the timeout to be null so it waits indefinitely. Now we store the incoming activity in an int variable called activity. We then check whether this variable is equal to SOCKET_ERROR, if that is so then we print an error code and exit with an exit failure command.

FD_ISSET(master, &readfds) : If something happened on the master socket, then there is an incoming connection

We check for the new incoming connections and if the connection is successful then we send a welcome message to the client. After this we add the client to the array of sockets.

Now we check for some i/o operation on some other socket.

If the client is present in read sockets than we get the details of the client by getpeername(s, (struct sockaddr)&address, (int)&addrlen);

Check if it was for closing, and also read the incoming message by valread = recv(s, buffer, MAX_BUFFER_SIZE, 0);

We have some cases for the value of valread

- **SOCKET_ERROR :** We get the error code and check for the cause of the error. We print the respective message.

- **0 :** Some client disconnected, so take that client information and print the details of the client and close the socket connected with that client.

- **else :** We echo back the incoming message in the buffer. we collect all the incoming data in an array of characters, and set the flag value for that client to be true with means that we have received the message from that client.

Once all the clients have sent the message then we concatenate the incoming strings from the clients and server. Now the new concatenated message is sent to all the clients and we set all the clients flag to false which means we need to wait until every one sends message again to the server and keep on repeating this in a loop.

2.9 Player header file (Player.h)

In this file we declare a struct which has a constructor for player (which sets the username, position, and character index accordingly), name, player index, and position of player.