Bubble Trouble Game Development in WebGL

Pedro Emanuel Amaral Santos
José Pedro Almeida Duarte

***Resumo* – O presente artigo apresenta as etapas de elaboração do conhecido jogo 2D "Bubble Trouble" numa versão 3D em WebGL. Pretende-se mostrar as várias potencialidades do WebGL na área da Computação Visual elaborando-as num exemplo concreto.**

Usando uma abordagem de descrição passo a passo da organização do jogo desenvolvido, este artigo aborda os conceitos básicos de WebGL e tenta, essencialmente, torná-la (esta ferramenta) de fácil compreensão para qualquer novo estudante da abrangente área de computação visual. Para além desta abordagem, este artigo irá também abordar aspetos de carácter específico e avançado de WebGL, proporcionando ao leitor uma melhor compreensão das potencialidade da Computação Visual combinada com WebGL.

***Abstract* – This paper presents the different phases concerning to the elaboration of the known 2D game "Bubble Trouble" into a 3D version developed in WebGL. It is intended to show the various potentialities of WebGL in what concerns to Computer Vision turn them explicit in a concrete example.**

Using a step-by-step description approach of the organization of the developed game, this article will address the basics of WebGL and it will try to make it clear to understand to any new computer vision student who want to learn this useful tool. Besides this approach, this article will also dive into some specific and advanced details of WebGL, providing to the reader a better understanding of what Computer Vision combined with WebGL is capable of.

I. INTRODUCTION

WebGL is a JavaScript tool, often thought as a 3D API, for rendering interactive 2D and 3D graphics into any web browser. It can be considered as a rasterization engine that draw primitives, such as points, lines and triangles, based on the code that the programmer supply. WebGL runs over the Graphic Process Unit what means that the appropriated code that runs on that unit has to be provided. This is accomplished in WebGL using the shaders' functions, vertex shader and fragment shader.

The fragment shader computes a color for each pixel of the primitive being drawn.

The vertex shader computes vertex positions and can rasterize various kinds of primitives including points, lines, or triangles.

WebGL only cares about clip space coordinates and color so it expects the programmer to provide these two things: a vertex shader that provides the clip space coordinates and a fragment shader that provides the color.

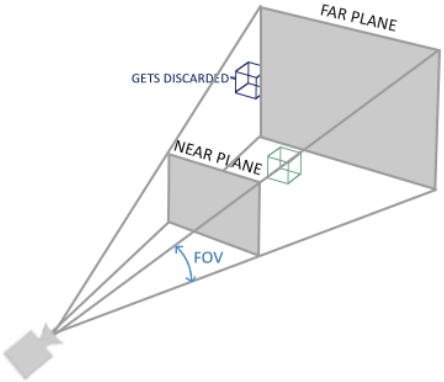

Fig. 1 - Representation of the clip space

It is important to understand the concept of clip space. Clip space is the area that can be seen by the observer, the area where the primitives have really to be draw because they will not be hidden.

The WebGL will be deeply discussed in the next chapter. For now, the Bubble Trouble game will be explained.

Bubble Trouble is a game that can be played by 1 or 2 players, a red devil and a blue devil. This game consists in having the players inside a square area that can move horizontally and are limited by the vertical walls at the end of each side. There are bubbles appearing and these bubbles bounce on terrain and if it touches the player, he loses. Players can shoot the bubbles but only vertically. When they hit it, they hit one, that bubble will decompose itself in two smaller ones. The purpose of the game is that the player avoid and shoot the bubbles until them disappear, what means that the player, or the players, have to shoot the bubbles so many times as it needed for the bubbles to be so small that they are not able to decompose no more.

The game can be played online the Miniclip web page. There is some Desktop versions that can be found around the internet, some of them, fan made ones, are different from the original in many ways but the basic concepts are still there.

Fig 2 – Original Bubble Trouble game

A particular aspect of our implementation, is that we decide to build our project based on the organization and base functions of the demo project of the webgl-obj-loader (which is mentioned in references) because it abstract us from interacting with third party programs and focus on the WebGL details. In this case, we wanted to import ".obj" files from Blender. We only used the base code of that project which is similar to the code used in classes. The reason for that was the outstanding code organization that made us to progress in WebGL learning far too much.

II. CREATING MODELS

We needed three models to develop this game in WebGL. A model for the bubbles, a model for the player, which is a tank, and a model for the world where the game happens. The model for the bubble was build and blender as like as the world. These models are exported in the ".obj" format which contains information relative to the geometric vertices, texture coordinates, vertex normal and other. The model of the tank was taken from the internet and it was imported through our project making use an adaptation of the webgl-object-loader. The main focus of this loader is to easily allow importing models without concerning about how another 3D graphics program work and built models, keeping it low-level enough so that the focus was on WebGL rather than another high-level program or framework. The webgl-object-loader will parse through the OBJ file data and collect the vertex, vertex normal, texture and face information of the model.

III. USING MODELS IN WEBGL

In order to use the models we make use of a drawObject function which takes as arguments the model that points to the mesh obtained from the webgl-obj-loader when parsing the ".obj" file, the shiness and the color of the object. This function is using while drawing the world, the drawWorld function, to draw not only the world itself but also the tank, the bubbles and the bombs.

IV. ORGANIZATION

In this section, the organization of the project you be presented in order to clarify how everything works and is connected. It will be presented the essential things in each file.

A. *Index*

This file contains the definition of all html, what means it displays the organization of the contents of the web page. In it, it is defined the the canvas used for the WebGL, functional buttons and information that will be explained in further topics in this article.

In this file it is also loaded all the dependencies, such as all the javascript files need to run the game. The shaders are also created in this file.

B. *glMatrix*

This file contains auxiliary methods to create and manipulate matrixes, such as rotations and translations, methods to manipulate vectors and other functionalities essential to help building the WebGL environment.

C. Globals

In this file all the global variables are defined in order to keep the state of the game and being able to the next state in function of the previous one. The way it is organized was crucial to make the development easy and scalable. All the state of the game is fully described and easy accessible and readable what makes it a lot more easy to implement more and more features. Some examples we keep track in this file are the current camera being used, the illumination factors, some game variables such as score, level and life, tank and bubbles information and some other WebGL variables.

D. webgl-loader-js

The webgl-loader-js creates and loads the mesh from the ".obj" files making it possible to easily read and use complex models made in third party programs our game.

E. web-gl-utils

This file, from Google, contains functions that every webgl program need. It creates the 3D context not makins us to create it manually. This avoid the presence of errors and checks if there is success or failure when creating the context.

F. Webgl

This file starts all the WebGL flow by loading up the model meshes and then calling the webGLStart functions. This last function will get the canvas previously defined in the html, initialize webGL in that canvas, initialize the shaders, initialize the buffers, initialize the textures and, at last, set the gl color to the initial one, black and enable the depth test. Some of the functions mentions are invoked from the file called initialize.js that explicitly initializes the gl variable, the canvas, the shaders and the buffers of the previously obtained meshes.

Backing to the webgl file, we have the function called "start" that activates the tick and calls the function with the same name that is responsible to update the scene's animation state, draw the scene and also for arranges itself to be called again in an appropriate time, because it has to be called regularly.

More precisely, the tick function is regularly updating the canvas, and calling the animate function which, as it name says, animate the models of the world. This means that it will update the movement of the bubbles, update the movement of the tank given the user's input and verify collisions. It is also responsible for updating values in the html according to the actual state of the game, such as the score, number of lives and the reached level.

G. worldDraw

This file is in charge of placing everything respecting to the game in place where it belongs and can be seen by the user/player, the canvas. It sets the viewport and perspective in order to give the user the best view and experience when playing the game.

In order to have different views of the world and put the models in the right place, the model view matrix is manipulated. There is important things to take in consideration to properly display de models in the right place. When drawing a scene, it is told to webGL to draw each thing in the current position at the current rotation. In order to do that, it is needed to manipulate the model view matrix to draw the model in the expected position and, therefore, being able to get back to the previous state. This is accomplished using the mvPushMatrix and mvPopMatrix methods. The mvPushMatrix puts the matrix onto a stack in order to store its state and then the mvPopMatrix get rid of the current matrix to get back in order to the previous state. Having a stack means that it is possible to store nested states, each of which manipulating the model view matrix and afterwards restoring it.

H. tankMoves

The tankMoves file takes into consideration the keyboard input from the user, when interacting with the left and right arrows, and reproduces the movement of the tank.

This movement has a accelaration and it is limited by the limits of the world, which are its walls.

I. KeyHandler

This file ensures the correct use of the keys to play the game. The functionalities provided are the movement, left or right, the vertical shooting, changing the camera and manipulating zoom. The movement of the tank is accomplished through the arrow keys, left and right, respectively. Changing camera is made through the 'C' keyboard key. Finally, the zoom is handled with the page up and page down button. The first one is used to zoom in and the last one to zoom out.

J. Bubbles

This file uses a lot of physics logics trying to make the world the most similar possible to the real world.

Among other things, it verifies collision between two bubbles, between a bubble and the tank and between bubbles the world. It is also responsible update the movement of the bubble after collision and update the movement of the bomb, which can be considered a smaller bomb.

The detectCollisionBubbles function is the one responsible to detect collisions between bubbles and update their movement. It makes use of the Axis-aligned Bounding Box method to do it. This method consists in not checking if the circles overlap each other but it guarantees that they are, at least, near. After being sure that they are close, it is precisely checked if the overlap exists or not. The overlap is calculated given the two radius of each bubble and the distance between the center. If the sum of both radius are greater than the distance between their center, then there is a collision. This method makes it way more efficient to detect collisions because it only runs the most precise and time spending function when the bubbles are really near each other. Although the most precise method is avoid to be called, it is also optimized so that there is no need to use square roots and use multiplications instead. The square rootis more computationally expensive than multiplications so, we remove the need of the square root by square the sum of both radius and compare it with the sum of the delta X and Y squares. This algorithm is shown in the following pseudo-code:

```
double deltaXSquare = A.x - B.x
deltaXSquare *= deltaXSquare
double detaYSquare = A.y - B.y
deltaYSquare *= deltaYSquare

double sumRSquare = A.radius + B.radius
sumRSquare *= sumRSquare

if(deltaXSquare + deltaYSquare <= sumRSquare)
    // bubbles are touching
```

The detectCollisionBomb function detects the intersection between the bomb and the other bubbles and it uses exactly the method described for the previous mentioned function.

The collisionBubbleTank function detects if there a collision between the bubbles and the tank and if there is, it decrements the player's life counter and removes the bubble that collided with the tank.

The getBubbleY function updates the vertical position of the bomb according to the physics movement equations. Bubbles have an vertical acceleration that tries to represent the gravity of Earth.

The getBubbleX function updates the horizontal position of the bomb also according to the physics movement equations but it keep keeps the same velocity.

The movebomb function updates the movement of the bomb that only increases vertically and disappears from the world if it reaches the top.

V. The Game

The final result presented is not totally equal to the bubble trouble but it is quite similar. Since it is not the same, in this chapter we will address our approach to game explaining the rules and some details of the implementation.

Starting with the player, instead of a red evil guy, there is a tank that also moves horizontally in the world. The world is nothing more than a rectangular prism open in one of the lateral sides (longer ones). The player can move only according to the XX axys and cannot move through the walls of the world. If the user presses the movement button continuously, the tank will get some acceleration.

Besides this, there are bubbles jumping around bouncing in the walls and in each other. The player has to avoid touching them and has to destroy them to gain points.

In order to destroy bubbles, the player can make use of his weapon that can only shoot in a vertical direction, what means, following the YY axys. Shooting makes him lose its velocity. When the bomb hits a bubble, that bubble disappears and the player's score is incremented by one.

If the player reaches a predifinied score, the level will increase and a large number of bubbles will be generated making the game harder as the level increases. If the player reaches a high level, also previously defined, he will be able to shoot at a higher fire rate. When a new level is reached the user can hear a sound indicating that event and can also observe that the color of the bubbles had been changed.

The player has 3 lives, what means three opportunities to being hitted by a bubble without losing the game.

It is possible to play using four different cameras and it is also possible to change the zoom of the image following the instructions provided on the border of the game.

There are also buttons to start and restart the game that are properly identified with the same name, according to their functions.

V. Final Considerations

There are some important aspects that need to be mentioned so that the game can be run and understood.

In order to run the game, because it is not available in a webpage still and some web browsers don't allow some content, such as ".obj" files, to be imported due to security reasons, it is needed for the user to install a http server and open our project in it. An easy way is to install nodejs and npm, install http-server through the command "npm install http-server" and then, after moving to the project folder, execute the command "http-server .", which will initialize the web server in the current directory. By now, the user will be able to access it in the web browser through the localhost address.

The game was also developed for a Full HD resolution (1920x1080) and can show some irregularities in other displays, more precisely in what concerns to the position of the HTML elements.

In what concerns to the individual work performed by the two authors, Pedro Santos produced around 55% of the final solution and José Duarte the remaining 45%.

V. Conclusions

There were a lot of difficulties to face and the learning curve had an exponential growth but it was hard to start developing the game and understand the fundamentals of webGL.

One of the main difficulties was to load ".obj" files into our project. Not making use of other frameworks, such as three.js, made us stuck in low-level programming for long time.

We had to research about game development physics in order to make the game movements and collisions the most real possible.

To conclude, this project granted us a lot more acquaintance to webGL and knowledge in other subjects related to Computer Vision.

REFERENCES

[1] WebGL Fundamentals. Available at WWW: <URL: https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html>.

[2] Webgl-obj-loader, GitHub, Aaron Boman. Available at WWW: <URL: https://github.com/frenchtoast747>.

[3] Learning WebGL, Giles. Available at WWW: <URL: http://learningwebgl.com/blog/>.

[4] Bubble Trouble, Wikipedia. Available at WWW: <URL: https://en.wikipedia.org/wiki/Bubble_Trouble_(1996_video_game) />.

[5] WebGL, Wikipedia. Available at WWW: <URL: https://en.wikipedia.org/wiki/WebGL />.

[6] When Worlds Collide: Simulating Circle-Circle Collisions. Darran Jamieson. Available at WWW: <URL: https://gamedevelopment.tutsplus.com/tutorials/when-worlds-collide-simulating-circle-circle-collisions--gamedev-769/>.