Bottle Shader and Texture Data Streaming

Anton Nilsson
antni004@student.liu.se

1. Introduction

In May 2020 the game Half-Life Alyx received an update which made, the previously opaque bottles, filled with realistic looking liquids. Although the effect looks convincing it is simply clever shader tricks and not real liquid simulation. This comes with some limitations to how the liquid can behave. However, this faked liquid is much faster than doing real liquid simulation and it is therefore better suited for real-time rendering in games. This project aimed to replicate some of the convincing effects of the liquid shader in Half-Life Alyx.

The, atleast partially, implemented mandatory and optional features are listed under Section 1.1 and Section 1.2, respectively. Other sections of the report include background information and theory, dependencies, implementation details and a conclusion, as well as a link to the source code.

1.1. Mandatory Features

- Transparency.
- Different bottle models.
- Refraction through the glass.
- Faked liquid surface using a plane and sinus waves.
- User input to apply forces.

1.2. Optional Features

- Forces through rotations.
- Animated textures.
- Foam.

2. Background Information

This section describes some background information and theory needed to understand some of the concepts in Section 3 and Section 4.

2.1. Bottle Volume

One important part of this project is volume preservation of the liquid inside of the bottle. This is explained further in Section 2.2 and Section 4.3. This section instead focus on two methods for calculating the volume of the entire model V_B.

2.1.1. Volume using Sum of Partial Cones

The bottle models used in this project are limited to being radially symmetrical around the y-axis. Since the bottles themselves are also made up of polygons, this comes with advantage that the volume of the entire bottle is easy to calculate using cones. Each section of a bottle is a partial cone with, a height, a starting radius and an ending radius. The volume of the entire bottle is given by Equation 1.

$$V_B = \sum_{V \in \text{ Cones}} V \tag{1.}$$

2.1.2. Partial Cone Volume from Geometric Formula

One way of calculating the volume of a cone is using the geometric formula for cones. This formula is given in Equation 2, where h is the height of the cone and r is the radius at the base of the cone.

$$V = \frac{\pi \cdot h \cdot r^2}{3} \tag{2.}$$

A partial cone instead has two radii, r_1 and r_2, the radius at the bottom and top of the partial cone respectively. The formula for partial cones is given in Equation 3, where h again is the height of the partial cone.

$$V = \frac{\pi \cdot h}{3} \cdot \left(r_1^2 + r_1 \cdot r_2 + r_2^2\right) \tag{3.}$$

2.1.3. Partial Cone Volume from Integrals

Another way of calculating the volume of a partial cone is using a double integral. The inner integral is the area of a slice of the bottle at a given height. The outer integral is the volume of the entire bottle given those areas.

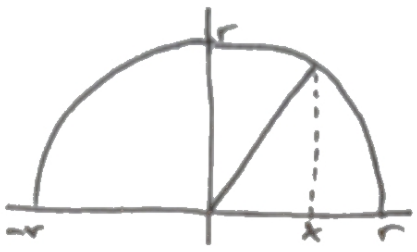

Figure 1: Height in circle from pythagorean theorem.

Since the bottle is radially symmetrical each slice of the bottle is a circle. This can be expressed using the pythagorean theorem. For a given radius, r and a distance from the center of the circle, x, the height is given by $2 \cdot \sqrt{r^2 - x^2}$. This is further visualized in Figure 1. The area of the circle is then given by the integral in Equation 4. The anti-derivative was calculated using an online tool[1].

$$\begin{aligned} A(r) &= 2 \cdot \int_{-r}^{r} \sqrt{r^2 - x^2}\,\mathrm{d}x = \left[r^2 \cdot \arcsin\left(\frac{x}{r}\right) + x \cdot \sqrt{r^2 - x^2}\right]_{-r}^{r} = \\ &= r^2 \cdot \left(\arcsin\left(\frac{r}{r}\right) - \arcsin\left(\frac{-r}{r}\right)\right) = \frac{\pi \cdot r^2}{2} \cdot (\operatorname{sign}(r) - \operatorname{sign}(-r)) = \pi \cdot r^2 \end{aligned} \tag{4.}$$

[1]https://www.integral-calculator.com/

The radius of the cone is a function of the distance from the bottom of the cone, y. This function is straight line equation given in Equation 5, where h is the height of the partial cone and r_1 and r_2 is the radius at the bottom and top of partial cone, respectively.

$$\begin{cases} r(y) = \frac{r_2 - r_1}{h} \cdot y + r_1 \\ r(0) = r_1 \\ r(h) = r_2 \\ r(y) = 0, \text{if } y < r_1 \text{ or } y > r_2 \end{cases} \tag{5.}$$

The integral for the volume of a partial cone is given in Equation 6.

$$\begin{aligned} V(h) &= \int_0^h A(r(y)) \,\mathrm{d}y = \pi \cdot \int_0^h r(y)^2 \,\mathrm{d}y = \pi \cdot \int_0^h \left(\frac{r_2 - r_1}{h} \cdot y + r_1\right)^2 \mathrm{d}y = \\ &= \pi \cdot \int_0^h \left(r_1^2 - 2 \cdot r_1 \cdot r_2 + r_2^2\right) \cdot \frac{y^2}{h^2} + \left(r_1 \cdot r_2 - r_1^2\right) \cdot \frac{2 \cdot y}{h} + r_1^2 \,\mathrm{d}y = \\ &= \pi \cdot \left[\left(r_1^2 - 2 \cdot r_1 \cdot r_2 + r_2^2\right) \cdot \frac{y^3}{3 \cdot h^2} + \left(r_1 \cdot r_2 - r_1^2\right) \cdot \frac{y^2}{h} + r_1^2 \cdot y\right]_0^h = \\ &= \frac{\pi \cdot h}{3} \cdot \left(r_1^2 + r_1 \cdot r_2 + r_2^2\right) \end{aligned} \tag{6.}$$

Now this is a lot of unnecessary math just to get to the same formula as the geometric one. However, this formulation has some advantages over the geometric definition. One advantage is that limits of the area integral can be changed. Our use case for this is explained in Section 2.2.

2.2. Liquid Volume

In order to preserve the volume of the liquid inside of the bottle, we need to be able to calculate the volume of the liquid. If the surface of the liquid is assumed to be a plane with the y-axis as it's normal vector, the result from Section 2.1.3 can be used. Equation 4 can be modified to only include the area less than a given value, the y coordinate, p, of our plane. This is shown in Equation 7.

$$\begin{aligned} A(r, p) &= 2 \cdot \int_{-r}^p \sqrt{r^2 - x^2} \,\mathrm{d}x = \left[r^2 \cdot \arcsin\left(\frac{x}{r}\right) + x \cdot \sqrt{r^2 - x^2}\right]_{-r}^p \\ &= r^2 \cdot \arcsin\left(\frac{p}{r}\right) + p \cdot \sqrt{r^2 - p^2} + \frac{\pi \cdot r^2}{2} \end{aligned} \tag{7.}$$

The bottle should also be able to rotate. However, Equation 7 requires that the center of the circle is at the origin of the coordinate system. By changing basis it can instead be the liquid plane that is rotating. This allows the integral to still be used. The value which is used to limit the area is then a function, $p(y)$, based on the rotation of the plane. The integral for the liquid volume using this change of basis is given by Equation 8.

$$V(h) = \int_0^h A(r(y), p(y)) \,\mathrm{d}y \tag{8.}$$

This volume integral could be expanded into an exact solution, but it results in a complicated expression with annoying function domains. This expression is not used in our implementation so it is not included here. This integral only calculates the volume of the liquid for a given partial cone of the bottle. The volume of the entire liquid contained inside of a bottle can be calculated using Equation 1.

2.3. Three Dimensional to Planar Rotation

To be able to perform the change of basis, the rotation of the bottle needs to be known. The rotation we are interested in should ignore the rotation around the y-axis. This can again be achieved by a change of basis. This new basis should be such that the rotation around the x-axis and z-axis in the old basis, is a rotation around just the z-axis of the new basis. We can get the new basis by using the rotation matrix of the bottle, R_B, on the y-axis basis vector v_y, by $v'_y = R_B \cdot v_y$.

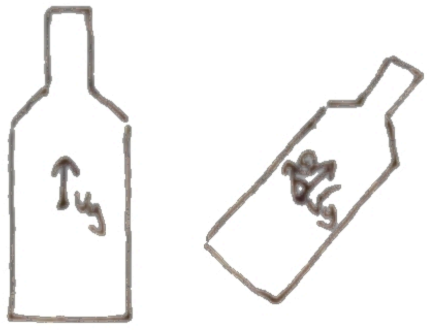

Figure 2: Vectors v_y and v'_y shown on a bottle.

This vector, v'_y is a linear combination of the y-axis and x-axis of the new basis. The rotation around the new z-axis is the same as the angle between the v_y and v'_y vectors. This angle, θ, can be calculated using Equation 9. This is also shown in Figure 2.

$$\theta = \arccos\left(\frac{v_y \bullet v'_y}{|v_y| \cdot |v'_y|}\right) \qquad 9.$$

2.4. Quaternions

Quaternions were introduced in the course book [1]. Their use case for video games is that they can be used to represent a rotation. A quaternion $q = \langle \cos(\theta/2), \sin(\theta/2) \cdot \boldsymbol{n} \rangle$, where $\boldsymbol{n} = \langle x, y, z \rangle$ is a unit vector, represents rotation around the $\boldsymbol{n}$-axis by θ radians. A quaternion $q = \langle w, \langle x, y, z \rangle \rangle$, can be converted to a rotation matrix using Equation 10.

$$R = \begin{pmatrix} 1 - 2 \cdot (y^2 + z^2) & 2 \cdot (x \cdot y - w \cdot z) & 2 \cdot (x \cdot z + w \cdot y) \\ 2 \cdot (x \cdot y + w \cdot z) & 1 - 2 \cdot (x^2 + z^2) & 2 \cdot (y \cdot z - w \cdot x) \\ 2 \cdot (x \cdot z - w \cdot y) & 2 \cdot (y \cdot z + w \cdot x) & 1 - 2 \cdot (x^2 + y^2) \end{pmatrix} \qquad 10.$$

Quaternions have some other nice properties such that they can be used to interpolate between different rotations. However, for this project they are simply used since it is a convenient way of representing rotation around an arbitrary axis.

2.5. Cubemaps

Cubemaps were introduced in the first course book [2]. A cubemap can be described as six different textures mapped onto the six sides of a cube, illustrated in Figure 3. This cube can be sampled using a three dimensional vector which maps onto one of the sides of the cube. This is commonly used for environment mappings and the course book gives the example that it can be used to fake reflections. This cubemap could also be used for refractions through transparent objects, which was the intended purpose for them in this project.

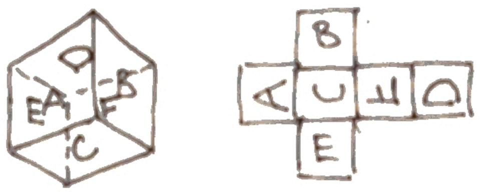

Figure 3: Cubemap.

2.6. Framebuffers

Another concept introduced in advanced course, course book [1] is framebuffer objects. A framebuffer object is a way of rendering to a texture instead of the viewport. For this project framebuffers are used to allow cubemaps to display a dynamic environment instead of a static one. Each frame, the scene is rendered with a 90 degree field of view for each side of the cubemap.

2.7. Binary Search

Binary search [3] is an algorithm which allows for finding a value meeting a specific criteria in a sorted set of values. It finds the desired value in $O(\log(n))$ time, by for each iteration dividing the search space in half, using divide and conquer. This is commonly used to find an index in a sorted list but our use case for this algorithm is explained in Section 4.3.

3. Dependencies

In addition to the header files used in the labs, the project also used some other external dependencies. These dependencies were, SDL, GamepadMotion and a NES emulator done as a hobby project.

3.1. SDL

Simple DirectMedia Layer or SDL[2] provides an unified API for interacting with operating system features. For this project it was used as a replacement for MicroGlut since it provides some features which MicroGlut does not. These features were for example access to the window position and inputs from a controller.

The structure of a SDL program is similar to a MicroGlut program. The `SDL_AppInit` function is similar to the `main` function of a normal `C` or `C++` program, but with no need to call a loop function, like `glutMainLoop`. `SDL_AppIterate` is like the display function of a MicroGlut program. The biggest difference between a MicroGlut program and a SDL program is that all events, eg. key presses, are explicitly sent to the `SDL_AppEvent` function where they can be handled.

[2]https://github.com/libsdl-org/SDL

3.2. GamepadMotion

GamepadMotion[3] is a header only library for sensor filtering and sensor fusion for gyroscope and accelerometer data. In this project it was used to allow the bottles to be rotated by physically rotating a gamepad with those sensors available. The gyroscope and accelerometer data is received from SDL and sent to a data struct each frame. This data gets turned into a rotation represented by a quaternion. This quaternion is converted into a rotation matrix as explained in Section 2.4.

3.3. NES Emulator

One of the goals of the project was that a dynamic scene would be refracted through the liquid inside of the bottle. The idea was that this would be in the form of a television screen which would have some kind of changing content on it. For this, a NES emulator done as a hobby project, was used.

For this project the important part of this emulator is how it gives a video output. The screen consists of pixels in a grid of a given width, W and height, H. A pixel is a struct of four, eight bit values, red, green, blue and alpha. Internally the grid is stored in two buffers. Each buffer is a $W \cdot H$ long array of these structs. In memory this is equivalent to a $4 \cdot W \cdot H$ long array of eight bit values, red, green, blue and alpha values alternating in that order. For each frame of the emulation one of these buffers receives the pixel values for the next frame while the other buffer contains the pixels currently being displayed, like double buffering in OpenGL. Another important detail of the screen representation is that the pixel stored at index zero is the pixel at the top left of the screen. Why this is important is explained in Section 5.2.

4. Implementation

This section explains the features of the project and the implementation of those features. The features explained are, user input for interacting with the bottle, volume preservation, the liquid shader and texture data streaming.

4.1. User Input

User input is used to allow the bottle to be rotated and shaken. How the bottle can be rotated has been described in Section 3.2. To apply forces on the liquid the position of the program window is used. This is a relatively simple calculation where the resulting velocity of the liquid is proportional to the change in position of the window. The code for handling user input is present in `source/main.cpp`.

[3] https://github.com/JibbSmart/GamepadMotionHelpers

4.2. Integral Approximation

As said in Section 2.2, it is possible to get an exact integral for calculating the volume of the liquid inside of a bottle. For this project it was instead decided that this integral should be approximated instead. For this approximation each cone is divided into a n by n grid, in this project $n = 128$. The width of the grid, W, is $W = \max(r_1, r_2)$, where r_1 and r_2 are the radii of the partial cone. The height of the grid, H, is the same as the height of the cone. Equation 11 then approximates the volume of liquid in the partial cone.

$$2 \cdot \frac{W}{n} \cdot \frac{H}{n} \cdot \sum_{\langle x,y \rangle \in X} \sqrt{r(y)^2 - x^2} \tag{11.}$$

This equation follows from what was explained in Section 2.1.3. The function $r(y)$ is the same as in Equation 5. The set X is the set of $\langle x, y \rangle$ coordinates in the grid which should be included in the approximation. Which these $\langle x, y \rangle$ coordinates are is expressed in Equation 12. This set is illustrated in Figure 4.

$$X = \{\langle x, y \rangle \mid |x| \leq r(y) \text{ and } \langle x, y \rangle \text{ is below the liquid plane}\} \tag{12.}$$

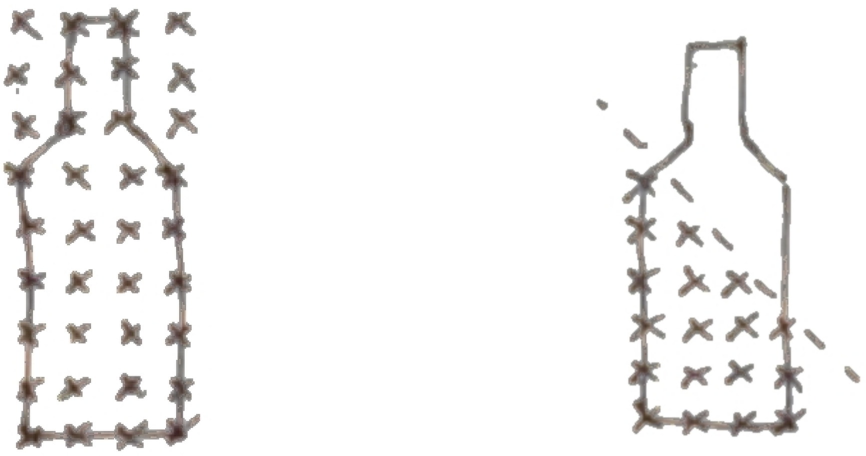

Figure 4: All points in grid (left) and points in the set (right)

To be able to determine which $\langle x, y \rangle$ coordinates are below the liquid plane we define a line. This line goes through the coordinate $\langle p_x, p_y \rangle$, which is a point in the plane, with a direction vector $\begin{pmatrix} \mathrm{d}x \\ \mathrm{d}y \end{pmatrix}$. To determine which side of the line the $\langle x, y \rangle$ coordinate is on we use the cross product as expressed in Equation 13.

$$\begin{pmatrix} \mathrm{d}x \\ \mathrm{d}y \end{pmatrix} \times \begin{pmatrix} x - p_x \\ y - p_y \end{pmatrix} > 0 \tag{13.}$$

4.3. Look Up Table

The liquid volume approximation described in Section 4.2 should now be used to find a y value which gives the targeted liquid volume given a rotation of the bottle. Since this value is unknown it has to be found somehow. For this we use binary search as described in Section 2.7. The search starts at the origin of the bottle model. If this gives a volume which is too low the y value is moved up by half of the radius of the bottle, otherwise it is moved down by half of the radius of the bottle. This increase or decrease is halved each iteration. This continues for a set amount of iterations, when the y value has been found.

Doing this search each frame is fine as long as the amount of bottles and iterations are small. However, these y values could instead be calculated before the program even starts. Given a set angles of the bottle and liquid volumes, a look up table could be generated. Using an image for this we could also take advantage of the ability to use linear interpolation when accessing texels on the GPU. The generated lookup table for a bottle is shown in Figure 5. The code for this table generation is located in `tools/bottle_volume.py`.

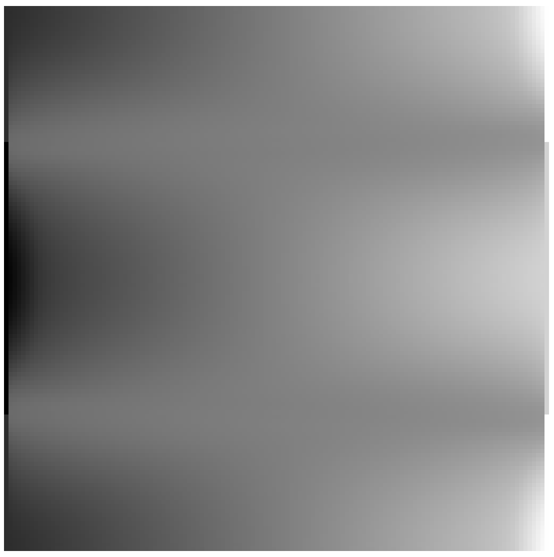

Figure 5: The lookup table for the bottle model used in the project.

This approach for liquid volume preservation is a simplification of volume preservation in containers which are not radially symmetrical. It is based on an unpublished paper from SAAB after discussions with one of the authors Axel Matstoms.

4.4. Liquid Shader

The bottle model is in reality two separate models, one larger outer model of the bottle and one smaller inner model of the bottle. The rendering of these models follows painter's algorithm as described in the first course book [2]. First the inner model is rendered as liquid, described in detail later in this section. Back-faces and front-faces of the models are then rendered as transparent as follows:

1. Back-face of the outer model.
2. Back-face of the inner model.
3. Front-face of the inner model.
4. Front-face of the outer model.
5. Repeat from 1. for each bottle in the scene sorted based on their z coordinate, in view coordinates.

The liquid is rendered by rendering both the front-face and back-face of the inner model. The back-face is drawn using the color we want the surface of the liquid to have and the front-face with some other darker color. Doing this makes it look like there is a modeled surface of the liquid, while in reality it is just the back-face of the inner model being used cleverly. This approach for how to replicate the Half-Life Alyx liquid shader was inspired by a fan implementation of the shader [4]. The rendering of both the bottle and the liquid is found in `source/objects/Bottle.cpp`.

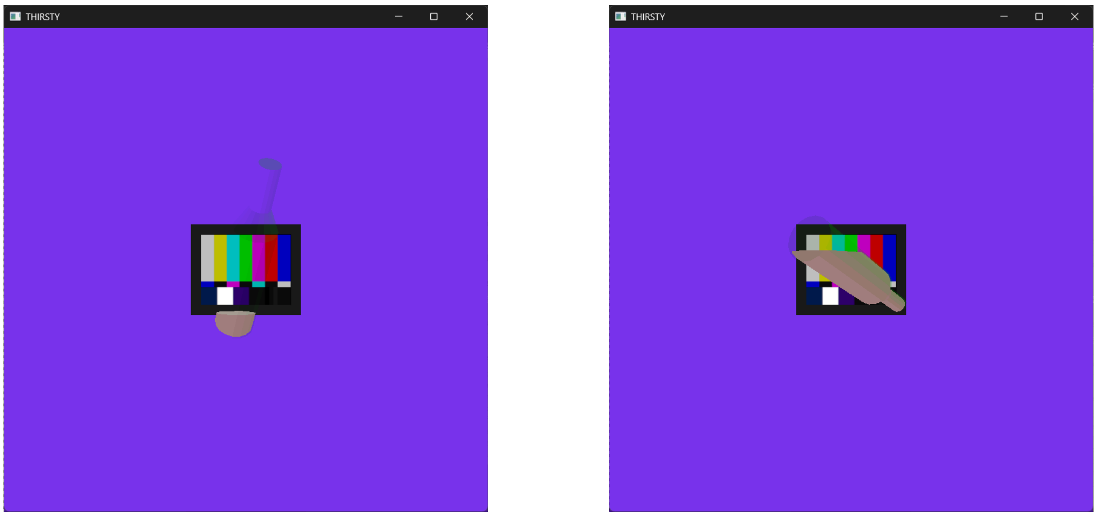

Figure 6: Two examples of bottles with different liquid volumes and rotations

The y value retrieved from the look up table described in Section 4.3 is passed from the vertex shader to the fragment shader. Inside the fragment shader there is a condition which checks if the current fragment is above this y level. Fragments which are above the found y level are discarded. This y level is also modified using a sinus curve based on the calculated velocity, described in Section 4.1. As the velocity of the liquid is decreasing foam is also added to the liquid based on the same y level. After a while the foam dissipates again. The shaders for both the glass and the liquid is found in the folder `shaders/bottle/`. Two examples of rotated bottles are shown in Figure 6.

4.5. Texture Data Streaming

When introducing framebuffers the course book talked about other ways of rendering the viewport to a texture [1]. The OpenGL function `glCopyTexSubImage` could be used to copy the content of the viewport to a texture. Since this requires that data is copied from one memory location to another, it was said that this should be avoided. However, when the data is on the CPU this copy operation is required, like in the case of the NES emulator from Section 3.3. For this OpenGL provides `glTexSubImage2D` or, in OpenGL 4.6, `glTextureSubImage2D` [5].

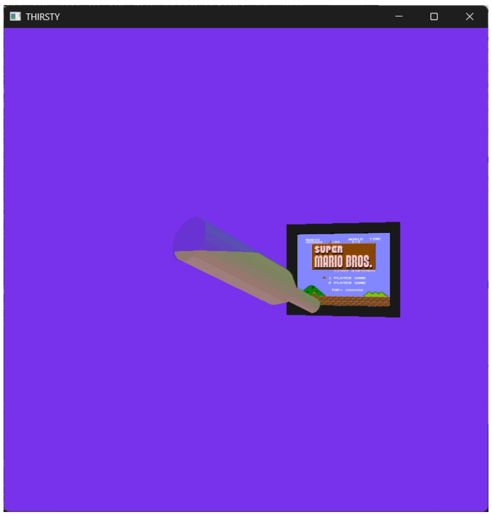

Figure 7: Streamed texture data displayed on the TV.

In this project this function is called each frame. This is fine since the size of the NES screen texture is *just* ~250kB of data and the rest of the scene is relatively simple. However, there are some ways in which this texture upload could be optimized further. The `glTextureSubImage2D` is a blocking operation, meaning that any operations which uses the texture need to finish before this operation can start. The first possible optimization is to only upload the new texture data when there is new data. This is faster as long as the framerate of the program is higher than that of the emulation. The second possible optimization is using multiple textures and then alternating between them. This is less likely to cause blockages since it is less likely that the texture being used to render is also being copied to. However, this comes at the cost of more memory usage on the GPU. The code for texture uploading is available in `source/GLScreen.tpp`. An example of an updated texture through data streaming is shown in Figure 7.

5. Problems

This section discuss two interesting problems encountered during the project. The two problems discussed are troubles getting refraction to work and the data order of texels in a texture.

5.1. Refraction

One of the listed features of the project was that the glass of the bottle should have refraction. The dynamic updating of the cubemap for the scene has been implemented as described in Section 2.6. However, when it was time to do the actual refraction it did not look as good as expected. The exact reason for this is unknown but one hypothesis is that the number of vertices in the model had a great effect on the end result. Unfortunately there are no examples of this ugly refraction, but the code for dynamic cubemaps is present in `source/CubemapRenderer.cpp`.

5.2. Texture Data Order

Another issue encountered during the project was when uploading texture data. As described in Section 3.3 the first pixel in the buffer is the top left pixel. The reason that this is a problem is that the OpenGL specification says that texture data is stored beginning from the lower left corner instead [6]. This resulted in the uploaded texture being drawn upside down as shown in Figure 8. This issue was resolved by flipping the texture coordinates when drawing the texture. This instead meant that the television test image, used as a placeholder, was drawn upside down. To fix this the image was edited to be upside down instead also shown in Figure 8.

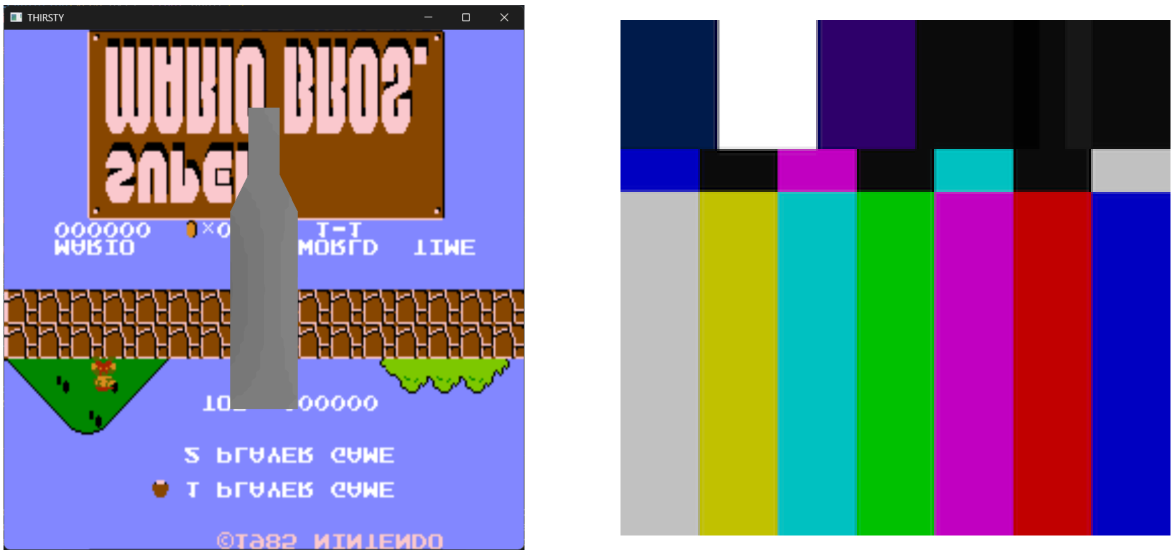

Figure 8: Uploaded texture being drawn down (left) and the flipped tv test screen (right)

6. Conclusion

In conclusion it is possible to make realistic looking bottles filled with liquid without having to do any actual liquid simulation. The issue of liquid preservation can be solved using integral approximation and look up tables. It is also possible to make the effect look even better by using sinus curves and foam build up. For steaming texture data to the GPU there are a lot of possible approaches, with different advantages.

Over all I am happy with how to project was able turn out despite the amount of time I was able to put into it. In the initial project plan actual liquid simulation was also included but was scratched at the half point. The preservation of liquid volume turned out to be a harder problem than I expected, since I found no information about it online. If I started the project again I would instead focus on doing liquid simulation from the start since I found more information about it.

7. Source Code

The source code for the projection is available in an open GitHub repository[4]. This repository contains a submodule for the NES emulator. The code relevant to this project from that submodule is included under Appendix A.

[4]https://github.com/antonegas/thirsty

References

[1] I. Ragnemalm, *So How Can You Make Them Scream?: Course book in advanced game programming*. Createspace Independent Publishing Platform, 2017.

[2] I. Ragnemalm, *Polygons Feel No Pain: Your Pocket-Friendly Companion When Learning Computer Graphics*. Createspace Independent Publishing Platform, 2024.

[3] CP-Algorithms, “Binary search.” Accessed: Dec. 30, 2025. [Online]. Available: https://cp-algorithms.com/num_methods/binary_search.html

[4] Minions Art, “Unity Liquid Shader (BIRP).” Accessed: Dec. 30, 2025. [Online]. Available: https://www.patreon.com/posts/unity-liquid-18245226

[5] Khronos Group, “glTexSubImage2D.” Accessed: Dec. 30, 2025. [Online]. Available: https://registry.khronos.org/OpenGL-Refpages/gl4/html/glTexSubImage2D.xhtml

[6] Khronos Group, “The OpenGL® Graphics System: A Specification.” 2022.

A. Screen Template Class

```
#ifndef H_SCREEN
#define H_SCREEN

#include <cstdint>
#include <array>

using std::uint8_t;
using std::size_t

template <size_t W, size_t H>
class Screen {
    public:
        virtual void put(size_t x, size_t y, uint8_t r, uint8_t g, uint8_t b);
        virtual void put(size_t x, size_t y, std::array<uint8_t, 3> color);
        virtual std::array<uint8_t, 3> get(size_t x, size_t y);
        virtual void swap();
    protected:
        typedef struct {
            uint8_t r = 0x00;
            uint8_t g = 0x00;
            uint8_t b = 0x00;
            uint8_t a = 0xFF;
        } RGBA;
        typedef std::array<RGBA, W * H> Buffer;

        std::array<Buffer, 2> buffers;
};

#endif // H_SCREEN
```