Alpas: Documentation

Uyvico, Valeriano, Capawing, Marces

December 2024

1 Description

Alpas is a retrowave-inspired game that takes on the idea of early 3D graphics on early computers. The name "Alpas" is inspired by the philosophy of breaking free — taking the limits of ASCII graphics and using it to provoke unimaginable experiences.

The game works behind the notion of the following concepts: infinity, freedom, and the future. It is infinite in a sense that it ends only when you hit an obstacle, otherwise you continue to progress endlessly. On the other hand, it targets the concept of freedom by finding a way to win in a seemingly-infinite game. Finally, the concept of the future is shown by using 3D projection on a console.

1.1 Game Mechanics

- Players start with an in-game menu featuring options to Resume, Restart, or Quit.
- Using keyboard controls, players navigate an animated car along an infinite road while avoiding collisions with oncoming obstacles.
- Players earn 1 point per object that the player overtakes. Level increases for point divisible by 4.
- However, avoiding collisions isn't the ultimate goal. Victory is achieved by escaping the road entirely, driving through fences to "break free." The game ends with a win only upon successful escape. Basically by spamming the left or right movement key 25+ times.

2 Technical & Mathematical Documentation

2.1 Technical Documentation

2.1.1 Build from Source

If you prefer to build from source, navigate to the directory of your computing environment (Mac/Windows).

```
cd windows
cd mac
```

Then run the following command(s).

For Mac:

```
rm alpas-main & rm alpas-keybinder & \
rm alpas-engine & make alpas-main && \
./alpas-main
```

For Windows, build with any compiler the following files:

```
alpas-main.c
```

2.1.2 Usage

From the compiled binary file, simply run it based on your operating system environment (Mac/Windows).

2.1.3 General Instructions

The game is fairly easy to play. We have the following masterlist for key mapping.

For main menu:

```
s: Start
x: Exit
h: Help
```

For game proper:

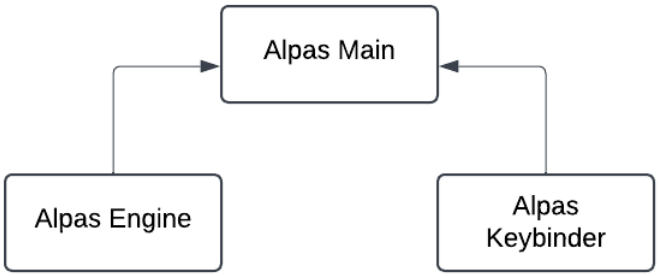
```
a: Left
d: Right
w: Forward
s: Backward
;: Move camera backward
p: Move camera forward
l: Lower camera
': Raise Camera
```

For game over menu:

r: Restart x: Exit m: Main Menu

2.1.4 Structure of Codebase

The codebase is divided into 3 main files.



The Alpas Engine contains most of the Mathematical Concepts described in section 2.2, such as Perspective Projection, and 3D translation.

Alpas Main on the other hand contains the majority of game logic, such as the winning conditions, collision, and other scoring logic.

Now the need for the Alpas keybinder lies in the reasoning of getting the program to work on both Mac and Windows, for ease of access the keybinder is on a separated file that would allow for variability in operating systems.

2.1.5 Documentation by Function

The following functions are the absolute crux of the project, if you want to dive deeper on the helper functions and other insignificant processes, comments on the actual code are placed throughout the source code (.c) files.

- renderScreen - responsible for rendering the screen buffer
- applyCameraTilt - applies rotation on the points based on a rotation matrix defined in 2.2.1
- project - Calculates projected 2D (x,y) ordered pair based on (x,y,z)
- drawLine - draw a straight line on a 2D screen using Bresenham's line algorithm.

- fillFace - fill the face of a 3D object using the Scanline Fill Algorithm
- translateBox - Translate object by a certain additive value
- calculateAABB - calculate Axis-Aligned Bounding box
- rotateVertices - apply rotation for any arbitrary object by editing the actual vertices in the memory
- performAnimations - performs animations of all objects (except obstacle)
- moveObstacles - moves the obstacle along the z and x-axis, checks collision for each time it is called
- game - main game loop calling all necessary helper functions

2.2 Mathematical Concepts

2.2.1 Rotation Matrix

The rotation matrix allows us to manipulate our viewing angle from a camera, using basic matrix multiplication we can transform our perspective.

Let R be a rotation matrix.

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{1}$$

On a 3D environment, let (x, y, z) be the position of the camera. Let C be a 3 x 1 matrix containing these positions.

$$C = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2}$$

We obtain the positions x', y', and z' which represent the transformed positions from CR.

$$CR = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{3}$$

This evaluates to:

$$CR = \begin{bmatrix} x \\ y \cdot cos(\theta) - z \cdot sin(\theta) \\ y \cdot sin(\theta) + z \cdot cos(\theta) \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{4}$$

This allows us to view from whichever angle we prefer—allowing us to have a seamless dynamic experience.

2.2.2 Perspective Projection

In computer vision, we have multiple ways to capture depth. One of which is how we calculate disparities (differences) from a binocular lens. The greater the disparity the farther the object, and vice versa. In perspective projection, we take that concept and sort-of inverse that process. This allows us to create the illusion of a 3D world existing beyond your computer screen.

Let z be the depth (distance from camera) and f the focal length of camera.

$$f = \frac{1}{tan(\frac{\theta}{2})} \text{ where } \theta \text{ is the FOV in radians} \tag{5}$$

Given the relationship of f and θ, we can actually use f as our FOV factor that we use for most projection formulae.

Now, given a point in the 3D plane, we can project this onto the 2D screen using the nature of the relationship of FOV and the actual 3D space. Hence,

$$x' = \frac{x}{-z} \cdot f \tag{6}$$

$$y' = \frac{y}{-z} \cdot f \tag{7}$$

2.2.3 Bresenham's Line Algorithm

Since we're dealing with vertices, edges, and faces from our original objects we need a way to display these seamlessly. We use Bresenham's Line Algorithm to assist with this. This process occurs after we've calculated x' and y' from Perspective Projection (see 2.1.2).

Given an initial point on the 2D screen (x_0, y_0) and a terminal point (x_1, y_1) we can draw a line by calculating the differences d_x and d_y between such points.

$$d_x = x_1 - x_0 \tag{8}$$
$$d_y = y_1 - y_0 \tag{9}$$

```
void drawLine(int x0, int y0, int x1, int y1) {
    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);

    int sx = (x0 < x1) ? 1 : -1; // Step for x
    int sy = (y0 < y1) ? 1 : -1; // Step for y
    int err = dx - dy; // Initial error term

    while (1) {
        // Plot the current point
        printf("(%d, %d)\n", x0, y0);

        // Break the loop if we reached the end point
        if (x0 == x1 && y0 == y1) {
            break;
        }

        int e2 = 2 * err;

        if (e2 > -dy) {
            err -= dy;
            x0 += sx; // Move in the x direction
        }
        if (e2 < dx) {
            err += dx;
            y0 += sy; // Move in the y direction
        }
    }
}
```

2.2.4 Scanline Fill Algorithm

Now that we've rendered our edges from the Bresenham's Line Algorithm, we now proceed to render the faces. We render the faces using the Scanline Fill Algorithm, the idea being that we iterate through each row of pixels and find intersections with that of the object's edges.

The x-coordinate of the intersection point along the scanline y can be found with the following derivation:

$$x = x_k + (y - y_k) \cdot \frac{x_m - x_k}{y_m - y_k}, \forall_{k,m} \in Z \tag{10}$$

We then render characters between the range of intersections.

2.2.5 Translation in 3D Space

For any object with vertex $V_{x,y,z} \in M$ where M is the set of all vertices defined on the object.

To allow for animations on the object, we translate such vertices by a certain additive numerical value d_k where k is any axis x, y, or z. We can translate V_k by the following formula,

$$V_k := V_k + d_k \tag{11}$$