Tyler's OPENGL "Physics" (lol) Documentation

TYLER SZETO

May 8, 2018

1 Abstract

In this paper I hope to illustrate my reasonings of how I created my quasi-physics engine in OpenGL. The hope is for the reader to be able to participate and modify my code so it actually works.

2 Preface

The "engine" I use is a modified version of "Spinning Cube" which can be found on OpenGL's website. [1]. As the reader can see, it was written in 1997. Finding C exclusive OpenGL projects is difficult, so this was the best I could find in the limited amount of time. As a result, I use it as a basis.

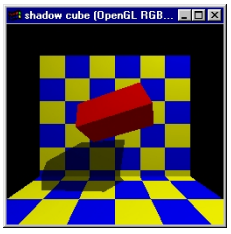

Figure 2.1: "Spinning Cube" from OpenGL's Website

[1] *https://www.opengl.org/archives/resources/code/samples/glut_examples/examples/examples.html*

In my code, I set the foreground and background to be green as to not have the checkered appearence. I also set the cube to be staying in an upright position rather than spinning, and increased the resolution of the project to a more modern display.
I will explain how to use my code in the following segment.

3 How to Understand and Use Code

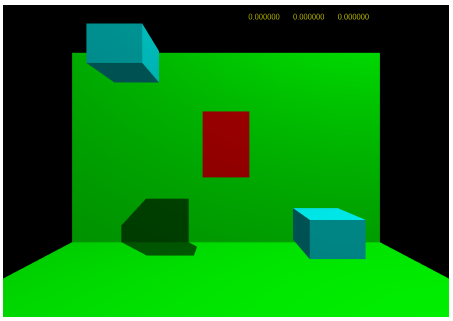

Figure 3.1: What my project looks like right now.

Figure **3.1** shows a contempary model of my project. In the top corner, coordinates of the red cube are displayed for the convience of debugging and working on hitbox detection. The red cube is the player controlled cube (later to be AI automatted), while the cyan cubes are static.

3.1 3D Coordinates

The cubes position is tracked with 3d coordinates, labeled **xPos, yPos,** and **zPos**. You can seem them displayed in the following Figure.

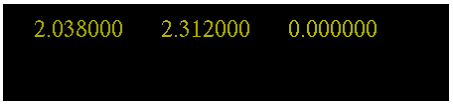

Figure 3.2: Coordinate Display.

For the time being zPos doesn't change, but depending on progress it may be something worth investigating. However, movement, collision detection are all handled using these coordinates, so it is important to understand how they function. We will focus on how coordinates are manipulated to achieve the afromentioned items in the following sections.

4 Movement

4.1 MOVEMENT - USER INPUT

There's two ways to handle movement so far in my system. The most default way is to right click anywhere on the screen, which props the following menu.

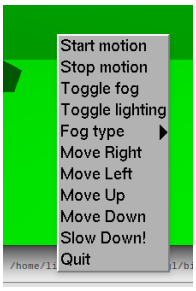

Figure 4.1: Movement Controls

This is useful for debugging. The SlowDown! command existed at one point to test the deceleration feature, but has since been disabled. The other commands are the remenents of the original "spinning cube" commands. This drop down feature is very useful for debugging!

```
void
menu_select(int mode)
{
  switch (mode) {
  ......
  //code
   .....
   .....
  case 7:
    horizontalVel = 0;
    if(moveDown)
        moveDown = 0;
    moveUp = !moveUp;
    break;
  case 8:
    horizontalVel  = 0;
    if(moveUp)
        moveUp = 0;
    moveDown = !moveDown;
    break;
  case 9:
    //slowDown = !slowDown;
    break;
  case 10:
    exit(0);
    break;
  }
  ...
  //later in program
  ....
```

```
  ....

  glutAddMenuEntry("Move Left", 6);
  glutAddMenuEntry("Move Up", 7);
  glutAddMenuEntry("Move Down", 8);
  glutAddMenuEntry("Slow Down!", 9);
  glutAddMenuEntry("Quit", 10);

  ....
}
```

It's very easy to add new commands to the menu, it is as simple as adding another case.
However, I have since then added movement with the traditional **"WASD"** setup (found in video games). The code can be found in the *"/* ARGSUSED1 */"* section, and is nearly identical to the switch statements.
Feel free to use *w*, *a*, *s*, *d* to move the cube around.

4.2 Movement- Cube Structure

```
struct cube{ //Red Cube Variables
double xPos;
double yPos;
double zPos;

int moveRight;
int moveLeft;
int moveUp;
int moveDown;

double verticalVel;
double horizontalVel;

double ACCELCONST;
double DECELCONST;

int slowDownHorizontal;
int slowDownVertical;
};
```

Briefly look over the variables being used in controlling the cube (the constants are subject to change). We store all these variables inside of a structure to allow for the quick passing around of the structure. The variables are pretty intuitive, and with the variables shown here, each integer is treated as a *boolean* variable essentially. Please see Figure 4.1 to see how these variables are being manipulated by user input.
We instantiate the redCube as a redCube object in the main class, which we pass around.

```
//...code....
struct cube *redCube;
//...code....
```

```
void initiliazeRedCube(){
redCube = malloc(sizeof(struct cube));

redCube->xPos = 0;
redCube->yPos =0;
redCube->zPos = 0;
redCube->moveRight = 0;
redCube->moveLeft = 0;
redCube->moveUp = 0;
redCube->moveDown = 0;
redCube->verticalVel = 0;
redCube->horizontalVel = 0;
redCube->ACCELCONST = 0.005;
redCube->DECELCONST = 0.5;
redCube->slowDownHorizontal = 0;
redCube->slowDownVertical = 0;
}
```

In order to modify redCube (i.e hitBox detection, movement, we need to send the entire structure over).

```
//part of main file..
static void controlMovement(){

  showCordinates(); //Show coordinate of player for debugging
  hitBoxDetection(redCube); //Enable hitbox detection
  handleMovement(redCube);
}

//part of movement.c

void handleMovement(struct cube *redCube){
  if(redCube->slowDownHorizontal && redCube->moveRight){
    redCube->verticalVel -= redCube->DECELCONST;
    if(redCube->verticalVel>=0){
    ......
```

4.3 Acceleration and Deceleration

Note that **"horizontalVelocity"** starts at zero. In other words, everytime this if statement is true, horizontalVelocity changes at a constant rate of **ACCELCONST**. This results in movement for the cube to be non-linear (before I think I sent a video of the movement being linear, and it hurt my brain to watch. It's really unappealing to look at.) I dont really remember why the variable of "modifier" exists though. I think it's safe to say we can remove it.
Whatever the value **Modifier** attains, we add it to xPos (going right, if we were going left we would be subtracting), which results in movement. Repeating my statement, as velocity as non-linear, the rate at which xPos changes is also non-linear.

```
  if(redCube.slowDownHorizontal && redCube.moveLeft){
    if(redCube.horizontalVel>=0){
        redCube.horizontalVel -= redCube.DECELCONST;
        double redCube.modifier = 0.4 * redCube.horizontalVel;
        redCube.xPos -= modifier;
    }
```

```
    if(redCube.horizontalVel<0){
    redCube.slowDownHorizontal = 0;
    redCube.moveLeft = 0;
    }
  }
```

We can see that deceleration is somewhat similiar to acceleration, but with a few more added steps.
Line 2:, we only continue to decelerate until verticalVel reaches zero. We do something similiar for deceleration similiar to acceleration, we continiously subject the constant of acceleration from the velocity, then update **xPos**.
Line 9: When **horizontalVel** is finally less than zero, we cease slowing down, thefore, we also cease moving in the left direction.
Since we now know how xPos and yPos are manipulated, we can see the code in which OpenGL moves the red box

```
  //Inside of a while loop
  ....
  code
  ...
  controlMovement();                    //See documentation!
  glTranslatef(xPos,yPos,zPos);
  glScalef(1.0, 2.0, .5);
  ....
  ...
```

4.4 "Collision Detection"

I put "Collision Detection" in quotations because it's very, very janky right now. I'm hoping ya'll can help but let's explain it anyways.

```
static void hitBoxDetection(){
  //BLOCK A HITBOXES

  if(moveRight && (xPos>2 && xPos < 4 && (yPos < -.5 && yPos > -10))){
    slowDownHorizontal = 1;
  }

  if(moveLeft && (xPos <8 && xPos > 4  && (yPos < -.5 && yPos > -10))){
    slowDownHorizontal = 1;
  }

  if(moveDown && (yPos<-1  && (xPos > 2 && xPos < 9.5))){
    slowDownVertical = 1;
  }

  //END BLOCK A HIT BOXES
 //  ....code for box B....
```

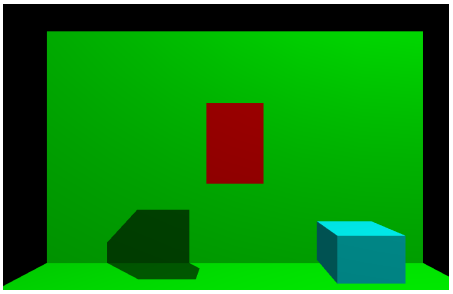

Figure 4.2: Showing box A and player only

Line 1: We see *moveRight* being assessed. Intuitively, this **if** case is only handeling the statement if we're currently moving to the right. The following statements are testing, if returned true, that means the cube is approaching the bounds in which I tell the cube to start slowing-Down, aka decelerating.

Note: Currently the values for hitbox detection are arbitarly and values that looked good for a demo. They need to be fixed still, so don't take them too seriosuly!

Imagine the field $x < 2 < 4$ and $y < -.5$. If either x or y values goes into that field, while going from the right, start to decelerate.
Line 5?: We see the **moveLeft** case now being accessed. The fundementals are similiar. I set the bounds to be 4 < x < 8 in this specific case in order to provide more deceleration room. I call this the buffer. I will discuss the buffer in the next segment shortly.

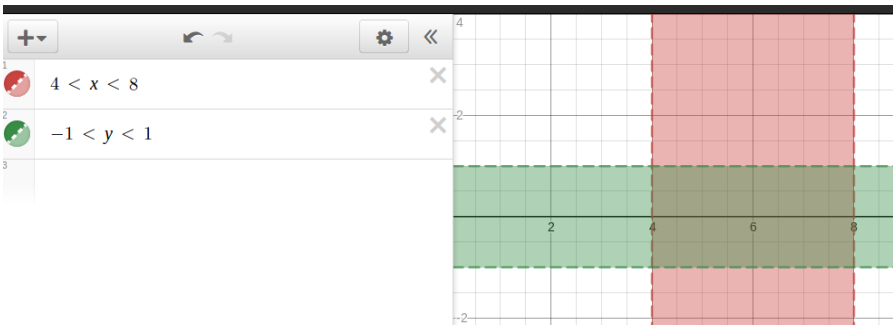

Figure 4.3: Plotting 4 < x < 8 and -1 < y < 1 in desmos. We can use this representation to represent areas where the cube cannot be, i.e, the collision. The code I posted above only has one "y" check, as opposed to the 2 listed in desmos, since cubeA is on the floor.

5 Graphics Rendering

```
drawCheck(6, 6, GREEN, GREEN);   /* draw ground */
```

```
  glPopMatrix();

  glPushMatrix();
  glTranslatef(0.0, 0.0, -0.9);
  glScalef(2.0, 2.0, 2.0);

  drawCheck(6, 6, GREY, BLUE);  /* draw back */
  glPopMatrix();

  //Custom Prop A
  drawCustomProp(0.2,0.2,0.2,4,-4,1,CYAN);

  //Custom Prop B
  drawCustomProp(0.2,0.2,0.2,-4,6,1,CYAN);
        .....
        .....

void drawCustomProp(double scaleX, double scaleY, double scaleZ,
                    double translateX, double translateY, double translateZ,
                    int COLOR){

                    glPushMatrix();
                    glScalef(scaleX,scaleY,scaleZ);
                    glTranslatef(translateX,translateY,translateZ);
                    drawCube(COLOR);
                    glPopMatrix();
  }
```

This is pretty self explanatory, I labled the variables pretty nicely. The *scale* variables (*scaleX, scaleY, scaleZ..* are in charge of how big the boxes will appear. The *translate* variables are in charge of where the box is placed. Lastly, COLOR is a defined integer (we can add more if we want) that will shade in the cube.
It's pretty easy to add in new props. Making the hitboxes is the harder part.

6 Please Help lol

I'll briefly outline what is not done and needs fixing.

6.1 AI

There's no actual AI. Everything is user controlled. We should figure out how to go about getting an AI to work, and how it should.

6.2 Better Physics

Consider the following code

```
  if(moveRight && (xPos>2 && xPos < 4 && (yPos < -.5 && yPos > -10))){
    slowDownHorizontal = 1;
```

I only gave the cube a small buffer to properly decelerate before hitting box A here. For slow enough speeds, the buffer is enough to prevent the box from colliding.

However, since movement in non linear, and the afromentioned code is a static buffer, if the cube is going fast enough, the buffer isn't enough to make it decelerate in time.

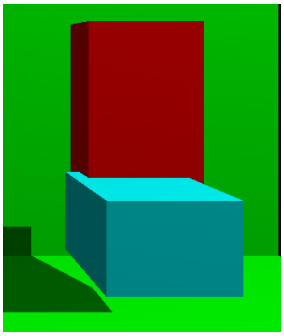

Figure 6.1: Collision oh no

Here's my idea.

```
  .....
  .....
double buffer = getDistanceNeeded();
if(moveRight && (xPos> (2-buffer) && xPos < 4 && yPos < -2 && yPos > 2){
  slowDownHorizontal = 1;
}
if(moveLeft && xPos <(4+buffer) && xPos > 2  && yPos < -2 && yPos > 2){
  slowDownHorizontal = 1;
}

 public static double getDistanceNeeded(....){
 //knowing distance from a coordinate, the current velocity, the
 //constants of deceleration, return how much distance is needed
 //to properly decelerate before hititng
 .......
 .......
 return distance;
 }
```

This way, the slowDownHorizontal proc can be set to one in an appropriate time, allowing the cube to stop properly, given a velocity without clipping through.

6.3 Bugs and Stuff

For movement, if you wish to cancel moving in a direction, press the key again. For example, if you are moving left, and wish to stop, press 'a' one more time, do not press 'd'. Don't press 'a' and 'w' at the same time either, it's too hard to actually control.
If the box is stopping next to a cube for some reason, that's a result of the janky hitboxes I hard coded in. Lets fix the hitboxes! Yeah!