Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Documentation for the Rubics cube

English

Brandt, Stephan

Project for the lecture Spielekonsoleprogrammierung

Betreuer: Prof. Dr. Christoph Luerig, B. Sc. Bartlomiej Pohl

Trier, 30.01.2017

Kurzfassung

In this document we'll explain the notions and different parts of my proyect for the lecture Spielekonsoleprogrammierung. The whole process of breaking ideas apart, researching and coding itself made it possible for me to create a programm for the PSVita in which we can manipulate and play with a 3D simulation of a rubics cube. This proyect was completly written on C++ with help of the SCE Library from Sony for the PSVita.

We'll go though my process of creation, what problems i encountered throughout the process and how i managed to solve them. We'll also talk about the mathematics behind the creation of a 3D game, rotation and logic in which I spend most of the time.
In the end, i'll briefly explain how the animation of the cube was made and some tips for the person who reads this documentation while going through the programm.

Inhaltsverzeichnis

1

Problem

1.1 How the development of the logic was a problem?

While developing the cube I encounter some complications throughout the initial implementation of the logic.

First I needed a logical representation of my cube in the abstract sense, which I would then convert to a graphical representation when needed, to draw the cube.

My initial idea of how to structure the Logical representation was to have a Cube object which had six face objects inside it, each face object had an array to store the nine colors of that face as well as a rotation value and a direction vector.

My graphical representation would have been an array which contained elements that held information of a position, and a color.

However as I was trying to implement the conversion from my logical to my graphical representation, I realized that it was going to be very difficult to rotate columns and rows. In the end I scraped this idea since I couldn't get my head around how to properly implement the rotation and the conversion from my object space into my world space was rather complicated.

2

Logic

2.1 How I managed to fix the logic

While exchanging ideas with a college, we realized that by just having a graphical representation, it was a cleaner and simpler logic to implement. The elements for the array would then contain a position and a color. Combining four elements would create a square of a certain color, combining nine squares would give a side of the cube, and combining six sides would give a cube in 3D space, giving me a total of four times nine time six elements to consider.

We realized that by just having a graphical representation, and manipulating it, we could easily construct our cube and let it be drawn. With a flag that each element possessed, we could change the behavior of that element, either rotating it in a certain way or not.

3

Structure

3.1 How I ended structuring my programm

In my program I only have two classes: The main.cpp and the Cube.cpp.

The Cube.cpp is where I create the representation of my cube, there I create the elements for the GPU to use. Building up the cube while at the same time filling our array with positions and colors.

In the class main.cpp, I used the example program from the lecture number 8, where the library and its components for rendering and the graphic allocation was already implemented and commented. Here I created the methods responsible for the rotation, transformation and manipulation of the cube, as well as handling inputs and updating animation.

4

Rubics cube

4.1 How is the cube created?

My Cube.cpp has two methods to create out the graphical representation:
One of this method is in charge of creating the small squares inside the faces of our cube, each small cube consists of four elements, with a total of nine cubes per face.

1. To create this elements we first get the direction in which the squares are facing, going from -1 to 1.
2. We iterate setting a small margin as separating line between our squares, we set an X, Y and Z component which describes a position in world space (since one of this axis is the one facing to us, we set this position as constant, and we only calculate new values for the other two).
3. A color value is assigned to each element to describe the color that should be shown on the screen, four elements which describe one square on one side should have the same color.
4. Each element also has a flag which I manipulate while the program is running, to determine if the point should be rotated or not.

With the method makeCube() we create the six faces of our cube and then store those elements inside our array of vertices that will represent our whole cube.

This class also creates a indices array which is used by the shader to identify which three elements create a triangle. Two triangles together creating a single square.

With another method we can set the flag of certain elements which I want to rotate. For example, if the side on the left of the cube should be rotated(from the perspective of the player), I set the rotation flag to true if the X component of an element is between -0.5 and -0.16. The points of my cube go from -0.5 to 0.5.

5

Rotation

5.1 How is the cube and it's faces rotated?

To rotate a certain side of the cube, first I will need to know along which axis I am rotating, then I need to know if its the first, second or third column/row. These values are saved in global variables. With this information I use a method checkSide() which selects a side given my information of which side should be rotated and sets the flag of those points in the side.

I then have to get a specific rotation matrix which includes the angle at which a side should be rotated and along which axis, which is described by the global variables.

When I finish rotating a certain side, I have a method which uses the rotation matrix of that side and manipulates my graphical representation by getting each point if the rotation flag I set, saving the X,Y and Z component in a Vector, multiplying the matrix onto that vector, and saving these new values back into out vertices array.

After I already have matrix to rotate a point, I send this information to the Vertex Shader via Uniform variables together with the points of my cube, and there I check which points need to be rotated. After this I apply a global rotation matrix to rotate all points on my cube equally, so that my cube can be rotated around itself and a player can see all sides of the cube.

I used quaternion mathematics to create the rotationmatrix of the entire cube because If I used normal matrices then my rotation would only be in 2 axis instead of three(because my joystick only has two axis). What I did was have a global quaternion which described my rotation of the whole cube which got adjusted by the joystick in each frame by multiplying a new quaternion onto the old one(the integration step), then normalizing it and converting it to a matrix.

6

Implementation of the touch and controller input

6.1 Touch inputs

How the touch is implemented is a fairly easy concept, we first create a report with the data acquired from the touch panel information. With this information we can find the position of a given finger touch applied on the panel and how many fingers were pressing the screen under other specifications, this two mentioned are the ones relevant for this solution.

First of all I only allowed one finger at a time to be utilized on the screen, after only one finger being used, we save this point as the origin point, and when we swipe throughout the screen, the last contact will be saved as the destination point. We then calculate the distance between this two points and create a vector out of it, and with the axis of this vector we interpolate with a quaternion and give that value to the update method to proceed to transform our cube.

The front panel wasn't implemented since I couldn't get my head around how ray casts work.
I didn't properly understand how to use normals on the surface hit by the ray cast, so I just scrapped the whole front touch idea and ended implementing a simple touch almost 1:1 to the back panel touch. But used to rotate a single side which was selected.

6.2 Controller inputs

I used the controls of the PS Vita as an alternative to not having touch on the front panel to change through faces and axis. Is also fairly simple, similar to the touch input, we get the controller report, and with it we can see which button was pressed, Then I changed the value of which axis should be rotated around by pressing either the triangle, or the square, or the circle. I selected the column/row, by using the arrow buttons left and right.

I used the Joystick data to rotate either the whole cube or a certain side. Since the joystick of the PSVita in its untouched state has a really small force applied to it, we first check if this force is big enough that it is obvious that a player is using it, and then we add the X and Y axis data to an accumulated angle used for the rotation. The left joystick is used for the transformation of the cube rotated with quaternions, and the right joystick is used to rotated the selected side with a normal matrix.

7

Animation and it's process

I implemented an animation when the player has stopped rotating a side so that this side gets snapped into its proper position, so that a different side can be rotated.

When the finger is released from the joystick or gets removed from the screen, a method gets called which calculated the angle to which the rotating side should get rotated to, these angles being increments of ninety degrees, I then keep rotating the side by adding a small factor to the matrix that is in charge of rotating that side, till the matrix describes a rotation equal to the angle that I want it to be at. When my side is rotated close enough, I change the values of the points of that side, in my vertex data structure.

8

Things to pay attention to while going through the code

There is a number of things that could be a problem for the reader while going through my code.

1. The nomenclature used for the names of the variables and methods could be confusing, I suggest to read the comments in the code to go along with the documentation, certain variables and methods have a name that don't necessarily explain their behavior properly.
2. Methods in the main.cpp are rather spread out, some of them are help methods to support calculations inside other methods.
3. The controllers to select and swap between the axis of a cube could be confusing for the player and needs some time to get used to.
4. The methods could have been separated in smaller and compact classes to make a more consistent and clearer code, in the current state is rather complicated to go through the classes.
5. There are a few global variables that have similar names.