Introduction to Visual Computing

Assignment# 3

User Interaction Design

March 5, 2019

Description

In this session you will be facing a real design problem: what are the possible interaction styles to manipulate a plate in 3D space, and which one provides the smoothest user experience?

Objectives

- Add interactivity to your last week's code.
- Use Processing 3D rendering mode and 3D built-in functions to create and transform a cuboid.
- Create user interaction that allow for rotating the cuboid.

Specific Challenges

Provide an intuitive user interaction with a 3D object on the screen, using only keyboard and mouse.

Preliminary steps

- If you have not completed assignment#2, finish the missing parts.
- Make a new copy of your project and rename it to "InteractiveProjections.pde".

Part I

Make your last week's code interactive

Make sure that you have done the preliminary step given in the first page. In this part you should make your application interactive, in the following way:

- The user drags the mouse up and down to scale the cuboid up and down. (`http://processing.org/reference/mouseDragged_`)
- The user uses UP and DOWN keys to rotate the cuboid around the X axis.(`http://processing.org/reference/keyCode`)
- The user uses RIGHT and LEFT keys to rotate the cuboid around the Y axis.

Try to analyze the user interaction within your group. Which parts do not seem intuitive/natural to you? How would you make it otherwise?

Part II

P3D

In this part you start using the Processing 3D rendering mode, and for the rest of this project you are allowed to use the 3D built-in functions. Firstly you need to set the rendering mode in the `size()` function.

```
void settings() {
  size(200, 200, P3D);
}
```

Taking it further (optional)

GPUs and graphics APIs like OpenGL have been moving away from the fixed-pipeline model to a fully programmable approach where every rendering operation needs to be coded as a shader program. Following this trend, in Processing 3, the P3D renderer implements all the drawing operations with GLSL shaders. A significant advantage of moving to programmable pipelines is the increased performance. The drawback, on the other hand, is that devices that don't support OpenGL 2.0 and programmable shaders won't be able to run P3D sketches.

Step 1 – Basic Shapes

To draw a cuboid or a sphere you should use the `box()` and `sphere()` functions, which have parameters to only specify their size. To position them in the space you should use transformation functions, and/or change the camera setting. The functions that are usually used for transformation (`translate()`, `rotate()`, `scale()`, `pushMatrix()`, `popMatrix()`) work almost identically in 2D and 3D. In 3D, instead of one `rotate()` function there are three `rotateX()`, `rotateY()`, `rotateZ()`.

To start, copy the following code and give it a try.

```
void settings() {
  size(500, 500, P3D);
}

void setup() {
  noStroke();
}
```

```
void draw() {
  background(200);
  translate(width/2, height/2, 0);
  rotateX(PI/8);
  rotateY(PI/8);
  box(100, 80, 60);
  translate(100, 0, 0);
  sphere(50);
}
```

Step 2 – Camera

All 3D rendering methods rely on a model for a scene (which you can change with the transformation functions) and a camera that observes it (or eye as you learned last week). The `camera()` function in Processing is derived from OpenGL, and is used to set the position and orientation of the camera. There are nine parameters, arranged in groups of three: (1) the position of the camera, (2) the location it is pointing to, and (3) its orientation (which axis looks up; the default being Y).

Run the given code below and test the effect of moving the camera. Try changing the other parameters of the `camera()` function and check the result.

```
void settings() {
  size(500, 500, P3D);
}

void setup() {
  noStroke();
}

void draw() {
  background(200);
  camera(mouseX, mouseY, 450, 250, 250, 0, 0, 1, 0);
  translate(width/2, height/2, 0);
  rotateX(PI/8);
  rotateY(PI/8);
  box(100, 80, 60);
  translate(100, 0, 0);
  sphere(50);
}
```

Note

If you don't set the camera, the following default parameters would be used:
`width/2, height/2, (height/2.0) / tan(PI*30 / 180),`
`width/2, height/2, 0,`
`0, 1, 0`

Step 3 – Lighting and Material

The lighting methods reveal the imagined dimension of 3D shapes. Lighting is turned off by default and enabled with one of the lighting functions: `lights()`, `ambientLight()`, `directionalLight()`, `pointLight()`, `spotLight()`. The following code uses `lights()` function, which adds some default lighting to the scene. For more controlled lighting possibilities you can check out the description of `ambientLight()` and `directionalLight()`, as they will become handy for the game design competition!

```
void settings() {
  size(500, 500, P3D);
}

void setup() {
  noStroke();
}

void draw() {
  background(200);
  lights();
  camera(mouseX, mouseY, 450, 250, 250, 0, 0, 1, 0);
  translate(width/2, height/2, 0);
  rotateX(PI/8);
  rotateY(PI/8);
  box(100, 80, 60);
  translate(100, 0, 0);
  sphere(50);
}
```

The quality of light interacts with the surface material. This is how you can create a mirror, for example. Processing offers a few functions to set the material: `shininess()`, `specular()`, `ambient()`, and `emissive()`. You should not worry about these for now. Nevertheless, you might want to employ them later to polish your game design.

Step 4 – Integration

The following code puts together the features that you have learned, and spices it up with some user interactivity. Try to understand how it works, you can reuse some of the techniques for the next parts.

```
float depth = 2000;

void settings() {
  size(500, 500, P3D);
}

void setup() {
  noStroke();
```

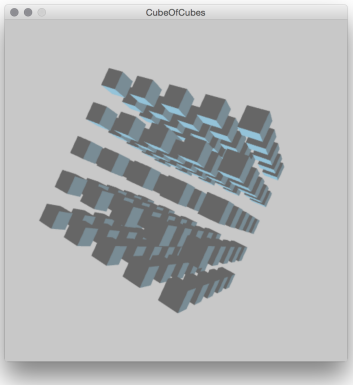
```
}

void draw() {
  camera(width/2, height/2, depth, 250, 250, 0, 0, 1, 0);
  directionalLight(50, 100, 125, 0, -1, 0);
  ambientLight(102, 102, 102);
  background(200);
  translate(width/2, height/2, 0);
  float ry = map(mouseY, 0, height, 0, PI);
  float rz = map(mouseX, 0, width, 0, PI);
  rotateZ(rz);
  rotateY(ry);
  for (int x = -2; x <= 2; x++) {
    for (int y = -2; y <= 2; y++) {
      for (int z = -2; z <= 2; z++) {
        pushMatrix();
        translate(100 * x, 100 * y, -100 * z);
        box(50);
        popMatrix();
      }
    }
  }
}

void keyPressed() {
  if (key == CODED) {
    if (keyCode == UP) {
      depth -= 50;
    }
    else if (keyCode == DOWN) {
      depth += 50;
    }
  }
}
```

Note

The `map()` function re-maps the value of its first parameter from one range to another. The original range is defined by the 2nd and 3rd parameters and the destination range by the 4th and 5th.


CubeOfCubes

Part III

Designing User Interaction

Please note that this is the main part of this assignment and is going to take significantly more time than the previous parts.

Create a new project and call it "Game.pde". This is going to be the project that you develop every week until reaching the final game application.

Firstly, you should draw a plate (a thin box), place it at the center of screen, and then provide the following user interactions:

- The user drags the mouse to tilt the plate around the X and Z axes, only up to 60 degrees.
- The user can change the speed of board movements, using the mouse wheel (`https://processing.org/reference/mouseWheel_`).

You can find a sample application on the course moodle page that shows these expected functions. **For the mac users:** you need to change the privacy setting to be able to run applications by unidentified developers. If the option does not appear in the Security and Privacy, see the guide here:
`http://osxdaily.com/2016/09/27/allow-apps-from-anywhere-macos-gatekeeper/`