Report for Project 4 - Patterns
CS 4497
Kamal Feracho

PART 1:

THE PROBLEM:

The goal is for students to implement two functions given vectors and points in which one function focuses on us having the figure “F” mirror itself given a vector parallel to some “U” vector or diagonal to it. Mirror symmetries ALWAYS pass through the center point which defaults it to the given O. The other function is an extension of the Mirror function called a Glide reflection which is simply a Mirror followed by a translation. The possible cases involves the center point in which an off-axis glide calls for the center point to change from the origin to “origin + V/4” in which the origin satisfies the condition for a normal axis Glide. For glide translations, there is ALWAYS a symmetry axis parallel to U. Implementing the two functions appropriately yields the outputs given in the provided picture folder.

MY APPROACH:

Unsuccessful Approach:

My first attempts involved replicating the formulas given in the slides and from the piazza post in hopes of providing the correct outputs. The first main formula I tried to implement was the following:

$$\mathcal{M}_{q,\underline{U}} \cdot p = p - 2(p - q) \cdot \underline{U}^{\perp}\underline{U}^{\perp}$$

From my understanding, p represent point P, q represents our V vector and U perpendicular is a vector thats rotated 90 degrees from the current axis of reflection vector which is then normalized. The U vector can be derived by finding the cross product of a random normalized vector and the given reflection axis vec normalized. Q represents the case for the glide axis which is C or C + V/4.

Glide reflection axis is either

- passing through rotation center C, or
- passing through the point $C + V/4$.

When evaluating the function, I found that “2(p-q)” reduces to a point, in which there was no present dot product function for points nd vectors. I tried hard coding it myself to see if it would work but had no luck. I have multiple failed attempts in implementing this function correctly as it yielded very fluky and incorrect results which led me to do more research. I have attached below screenshot of my coded attempts. (I would love to know what I was doing wrong).

Attempt 1:

```
/*
//eq.2: p = p - (2(p - q) • U)*U (othogonal vecs UU <---)
//pt P(float s, pt A) {return new pt(s*A.x,s*A.y,s*A.z); };                                    // sA
//pt A(pt A, pt B) {return new pt(A.x+B.x,A.y+B.y,A.z+B.z); };                                // A+B
//pt P(pt P, float s, vec V) {return new pt(P.x+s*V.x,P.y+s*V.y,P.z+s*V.z);}   // P+sV
//p = reflection_axis_vec
//P =p,
//vec N(vec U, vec V) {return V( U.y*V.z-U.z*V.y, U.z*V.x-U.x*V.z, U.x*V.y-U.y*V.x); };   // UxV cross product (normal to both)
vec UU = reflection_axis_vec.normalize();
vec norms = new vec(1,0,0);
norms = norms.normalize();
vec U = N(UU, norms);
// vec s1 = Va(d(reflection_axis_vec, n)* 2, reflection_axis_vec);
pt p = P;
pt O = WP.O;
// vec M(vec V) {return V(-V.x,-V.y,-V.z);};                                                  // -V
float a = d(Va(2, reflection_axis_vec), U);
//vec s1 = Va(d(Va(2, reflection_axis_vec), U)* 2, reflection_axis_vec);
U = Va(a, U);
//vec Va(float s, vec A) {return new vec(s*A.x,s*A.y,s*A.z); };                                 // sA
//vec V(pt P, pt Q) {return new vec(Q.x-P.x,Q.y-P.y,Q.z-P.z);};                               // PQ
//pt P(float s, pt A) {return new pt(s*A.x,s*A.y,s*A.z); };                                   // sA
//pt q = p;

// vec n = U(P, reflection_axis_pt);
//n.normalize();
//pt q = P(reflection_axis_pt, 1/4, reflection_axis_vec);
//q = P(-2, q);
*/
```

Attempt 2:

```
/*

vec UU = reflection_axis_vec.normalize();
vec norms = new vec(0,1,0);
norms = norms.normalize();
vec U = N(UU, norms);
pt p = P;
pt q = reflection_axis_pt;
'/eq.2: p = p - (2(p - q) • U)*U (othogonal vecs UU <---)

vec d = V(P, q);
float ddd = d(Va(2, d), U);
//float ddd = reflection_axis_vec.x*U.x + reflection_axis_vec.y*U.y + reflection_axis_vec.z+U.z;
vec zzz = Va(ddd, U);
//zzz.normalize();
P.x -= zzz.x;
P.y -= zzz.y;
P.z -= zzz.z;
   */
```

Correct Approach (Close enough)

I stumbled across this formula on stackexchange.com for reflection vectors, which worked very much in my favor for most cases except for case 8, which is only so slightly off. The formula was simple, $r=d-2(d\cdot n)n$

Where d = the given reflection vector, and n is some PQ/||PQ|| that has been normalized where P = point P and Q = reflection axis point.

https://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector

```
//eq: r = d - 2(d • n) * n where (d • n) is the dot roduct and n is normalized
//vec M(vec U, vec V) {return V(U.x-V.x,U.y-V.y,U.z-V.z);};
//vec U(pt P, pt Q) {return U(V(P,Q));};                                    // PQ/||PQ||
vec n = U(P, reflection_axis_pt);
// n.normalize();
// vec Va(float s, vec A) {return new vec(s*A.x,s*A.y,s*A.z); };             // sA
vec s1 = Va(d(reflection_axis_vec, n)* 2, reflection_axis_vec);
vec s2 = M(reflection_axis_vec, s1);
float x = s2.x;
float y = s2.y;
float z = s2.z;
P.x -= x;
P.y -= y;
P.z -= z;
```

After implementing that equation, the Glide method was very easy as it was a variation of the Mirror method but with an additional translation given some distance. The only additional thing we need to do is initialize a normalized glide axis vector that has been scaled by the given distance, then followed by a translation with the reflected point from the mirror function and the vector we scaled.

```
public pt Glide(pt P, pt glide_axis_pt, vec glide_axis_vec, float dist)
{
  //pt P(pt P, float s, vec V) {return new pt(P.x+s*V.x,P.y+s*V.y,P.z+s*V.z);}          // P+sV
  pt g = Mirror(P, glide_axis_pt, V(0,1,0));
  //P = P(g, dist, glide_axis_vec);'
  vec norms = glide_axis_vec.normalize();
  vec trans = Va(dist, norms);

  pt glide = Translate(g, trans);
 // P.z+= dist;
 // P.y+= dist;
  //P.x+= dist;
  //pt q = P(glide_axis_pt, 1/4, glide_axis_vec);
  //P = P(g, glide_axis_vec);
//  P.x += (g.x - dist);
  //  P.y += (g.y - dist);
    //  P.z += (g.z - dist);

  return glide;
}
```

Implementing the wallpaper cases was pretty straightforward. All we had to account for were all of the constraints for each Mirror/Glide transformation type and the respe

WPG Constraints on mirror symmetry

Mirror symmetry axis always passes through the rotation center C.

Mirror symmetry axis is either

- parallel to U vector or
- parallel to the diagonal (U-V).

WPG Constraints on mirror symmetry

Mirror symmetry axis always passes through the rotation center C.

Mirror symmetry axis is either

- parallel to U vector or
- parallel to the diagonal (U-V).

ctive translation formula for the distance.
Each example video can be found in the VIDEO folder.
“F” Demo is the name of the main demo for part 1.

Example “F”:

PART 2:

For the second part I implemented various classes that I previously had from previous course work and personal projects (of course, tweaked to accommodate the provided code). Some are functions put together from various classes I had to produce things like heads, rainbows, Suns, etc. Some are cool to see while others like the rainbow and mouse-pressed kaleidoscope implementation present a nice aesthetic apeal thats pleasing to the eyes and creates intricate patterns while following the same (or as close to for a couple of cases) transformation scheme as letter F. I will post a compilation video of all the patterns and implemented kaleidoscope variations. I have also implemented a special case 10 which consist of four 3D modes and a few of special modes. I’ve added a function to map the other functions that creates the different modes into an aesthetic polygonal/circular pattern around itself which creates crazy effects. Due to the amount of 3D objects being created, it is hard to see some of them which can

be helped with parameter tuning or a very fast processor. I've modified the gui accordingly to allow nice toggling through each of the modes and their nested modes (when applicable). I will post the code of the updated gui portion for easy navigation. The code for each mode is labeled respectively. There are classes created for each of the special 3D modes (Icosahedron, Cone) and the rose curve mode for easy use. I used these classes in another personal project hence why they have a similar set up when instantiating and calling them to draw. The final implementation is a texture mode which can be altered using the GUI. The texture modes is applied to all the 3D shapes that can be instantiated using PShapes and vertices calls, which allows for texturizing these classes. The textures tend to be blurry but is still a visible and noticeable change indicating it works. I will follow up the updated GUI photo with several video demo titles and their description which can be found in the VIDEO folder.

To summarize my part 2, there have been 23 different modes to change through, in which 20 are unique. There is one working texture mode which is when the shape created "Planets" are called. There is an incidengt of one of the sphere being texturized but also having a green hue. There is only one instance of this and all the other textures appear perfectly. I could not figure out what the main issue was in this case.

UPDATED GUI:

```
// ********************************************** TOGGLE THROUGH NORMAL MODES 0-10 **********************************
if(key=='1')
{
  mode = (mode +1)%11;
}

// ********************************************** TOGGLE THROUGH NESTED KALEIDOSCOPE MODES **********************************
if(key=='2')
{
  kmode = (kmode +1)%6;
}

// ********************************************** TOGGLE THROUGH NESTED 3D AND SPECIAL MODES **********************************
 if(key=='3')
{
  threeDm = (threeDm +1)%9;
}

// ********************************************** CHANGE DIRECTLY TO KALEDOSCOPE MODE **********************************
 if(key=='4')
{
  mode = 2;
}

// ********************************************** CHANGE DIRECTLY TO 3D AND SPECIAL MODE **********************************
if(key=='5')
{
  mode = 10;
}
```

```
   // ********************************************** Toggle Textures ***********************************
if(key=='t')
{
  if (texture2) {
    texture2 = !texture2;
  }
  texture = !texture;
}
     // ********************************************** Randomize Textures ***********************************
if(key=='6')
{
plans = int(random(100));

}
// ********************************************** Activates second set of Textures ***********************************

if(key=='p') {
  if (texture) {
    texture = !texture;
  }
  texture2 = !texture2;
}

// ********************************************** Randomize second set of Textures2 ***********************************
if(key=='7') {

 text = int(random(29));
}
```

“**Modes 0-10**” - demos base modes 0-10
“**Kali Modes**” - demos base modes of the Kaleidoscope
“**3D+Spec Modes**” - demos the 3D and special modes (better to see live, but can still be laggy at times)
“Textures Showcase” - demos the textures implementation