“I Believe I Can Fly”: An Exploration of 3D Boids

Sudhan Chitgopkar[1] and Michael Zhan[1]

[1]Harvard University, School of Engineering & Applied Science

1 Introduction

Originally developed by Craig Reynolds in 1986, “Boids” are an artificial life form which simulates flocking behavior in animals like birds or fish[1]. They have been the subject of interest of many applications in evolutionary computation, artificial life, and computer graphics due to their relatively straightforward implementation process and the organic visuals they can create. Like most other physical simulations, each “boid” (short for “bird-oid” object) has a position, velocity, and acceleration, which allows it to “fly” around a space. Additionally, boids have the ability to “view” other boids close to them and react to their position, velocity, and acceleration. Classically, this is done through three (3) rules, (1) separation, (2) alignment, and (3) cohesion, which are summarized in Figure 1.

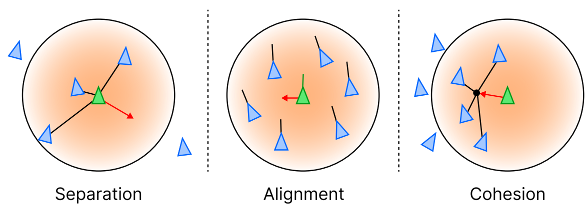


Figure 1: Canonical Boid Rules

Here, we use Processing[2] to implement a 3D boid simulation, with additional features to the canonical boids model including interactive rotation, material and color shaders, wall/obstacle avoidance, and real-time adjustable boid parameters.

2 Implementation

2.1 Design

This application is separated into five (5) separate classes, enumerated below:

1. `BoidRunner`: The main class, which handles running the `Boid` simulation.
2. `Boid`: A class which defines a `Boid` and its behavior.
3. `Obstacle`: A class which defines an `Obstacle` and its behavior.
4. `Arcball`: A class which defines an `Arcball` used to handle rotations.
5. `Quat`: A class which defines a `Quat`, used for the `Arcball` implementation.

[1]Reynolds, C. W. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model, in Computer Graphics, 21(4) (SIGGRAPH ’87 Conference Proceedings) pages 25-34.

[2]https://processing.org/

Additionally, the program uses the keybinds and mouse interactivity listed in the Table 1.

Key	Action
`x`	Toggle rotation around the x-axis.
`y`	Toggle rotation around the y-axis.
`z`	Toggle rotation around the z-axis.
`r`	Reset the program entirely.
`g`	Toggle GUI.
(`shift`) + `left click`	Rotate the scene. Use `shift` if GUI is visible, left click only otherwise.

Table 1: Simulation Keybindings

2.2 Flocking Environment

To get started, we need an environment for our Boids to flock in. Because this is a part of the simulation, it will be defined and drawn in `BoidRunner` but its properties (i.e. width) must be accessible by each `Boid` to ensure that the `Boid` remains within the environment's confines. Thus, let's define a size for the environment, which we'll henceforth just refer to as the SimBox:

```
final int BOX_WIDTH = 450;
```

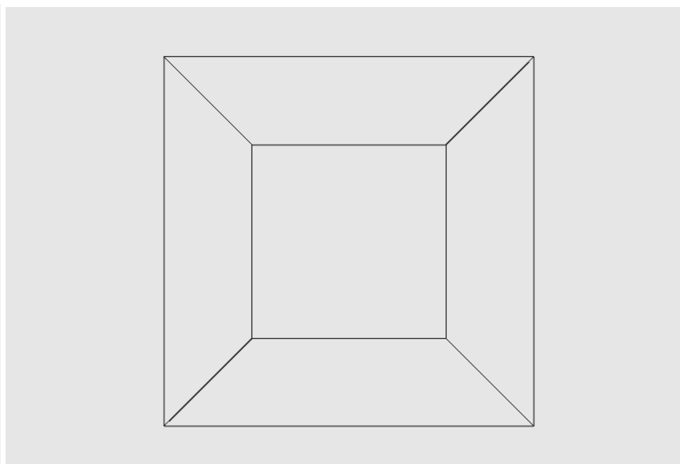
Now that we've defined the dimension of our SimBox, let's draw it to the screen.

```
final int BOX_WIDTH = 450;

void setup() {
    fullScreen(P3D); //required for 3D rendering
}
...
void draw() {
  background(255);  //clear frame between draw calls

  pushMatrix();
  translate(width/2, height/2, 0);

  noFill();
  stroke(0);        //white
  box(BOX_WIDTH);   //draw box
  popMatrix();
} //draw
```



While this is a good start, we're currently only presented with a single view angle, so it's unclear that this SimBox is three-dimensional. Ideally, we'd like to be able to rotate our scene so that we can see our `Boids` from all angles. If we draw all `Boids` relative to the SimBox, rotating the SimBox is sufficient to rotate all objects in the scene.

We can accomplish rotation using quaternions and an invisible arcball. This simplifies the mathematics required for correct rotation behavior and prevents roll. We implement quaternions as objects with four attributes: a scalar `w` and a vector comprised of `x`, `y`, and `z`, as well as with common methods such as multiplication and normalization. Notably, we track one main quaternion, `rotQuat`, that describes the current rotation state of SimBox at all times.

```
class Arcball {
    ...
    private PVector getBallQuat(PVector mouse) {
        PVector v = new PVector();
        v.x = mouse.x - center.x;
        v.y = mouse.y - center.y;

        float mag = v.x * v.x + v.y * v.y;
        v.z = (mag > radius_ * radius_) ? 0
              : Math.sqrt(radius_ * radius_ - mag));
        v.normalize();
        return v;
    } //getBallQuat

    public void update() {
        PVector start = getBallQuat(new PVector(pmouseX, pmouseY));
        PVector end = getBallQuat(new PVector(mouseX, mouseY));
        quat_ = mult(new Quat(0, end), mult(new Quat(0, start), -1));
    } //update
} //Arcball
```

We can implement the arcball as an `Arcball` class, which contains a center coordinate, radius, and a quaternion holding any new rotations made. Because the SimBox is not translated, the center remains in the center coordinates of the screen at an arbitrary depth of 0. To allow for rotation from anywhere on the screen, we let the arcball radius be equal to half of the larger window dimension. On any mouse click and drag, we call `Arcball.update()` to determine the vectors before and after the movement and then we calculate the corresponding quaternion. This quaternion is multiplied to `rotQuat` to modify the rotation of the actual SimBox.

To render the rotated SimBox, the built-in `rotate()` function in Processing is extremely convenient, taking in an axis and an angle of rotation. We can calculate the axis and angle from `rotQuat` and correctly apply rotation to the scene. This rotation is calculated every frame, and any change to `rotQuat` via the arcball is immediately shown in the rendered image. A partial implementation is shown below.

```
class Quat {
    ...
    public PVector getAxis() {
        float beta = Math.sqrt(1 - w_ * w_);
        return (beta < 0.0001) ? new PVector(1, 0, 0)
            : new PVector(x_ / beta, y_ / beta, z_ / beta);
    } //getAxis

    public float getAngle() {
        return Math.acos(w_) * 2;
    } //getAngle
} //Quat
```

```
void draw() {
    ...
    // draw rotated SimBox
    quatAngle = rotQuat.getAngle();
    quatAxis = rotQuat.getAxis();
    rotate(quatAngle, quatAxis.x, quatAxis.y, quatAxis.z);
    ...
    // update rotation quaternion based on arcball
    arcball.update();
    rotQuat = mult(arcball.getQuat(), rotQuat);
    ...
} //draw
```

In addition to manual mouse-based rotation, we can also develop animated rotations. Specifically, we choose to develop rotations around the x, y, and z axes relative to the camera frame. We define three quaternions specifically for this purpose, each from some small angle with a vector equal to one of the three basis vectors of the camera frame. Then, after normalizing these quaternions, we can multiply each to `rotQuat` to simulate a rotation around one of the three axes. By performing this multiplication and updating `rotQuat` every frame, we can animate the rotation of the SimBox. Lastly, we use the length-3 boolean array `xyzRotating` to control this behavior.

```
void draw() {
    ...
    // Define x, y, and z rotation quaternions
    xyzQuats[0].set((float) Math.cos(radians(0.2)), new PVector(-(float) Math.sin(radians(0.2)), 0, 0));
    xyzQuats[1].set((float) Math.cos(radians(0.2)), new PVector(0, (float) Math.sin(radians(0.2)), 0));
    xyzQuats[2].set((float) Math.cos(radians(0.2)), new PVector(0, 0, (float) -Math.sin(radians(0.2))));
    ...
    // Apply rotation quaternions every frame
    if (xyzRotating[0]) rotQuat = mult(xyzQuats[0], rotQuat);
    if (xyzRotating[1]) rotQuat = mult(xyzQuats[1], rotQuat);
    if (xyzRotating[2]) rotQuat = mult(xyzQuats[2], rotQuat);
    ...
} //draw
```

2.3 Flocking Behavior

2.3.1 Boid Setup

Now that we have a working environment for our Boids, let's begin creating Boids by creating the `Boid` class. For now, each `Boid` just needs to know the width of the SimBox (so as not to move outside of it) and information about the other `Boid`s in the simulation. We can pass these into the constructor as follows;

```
class Boid {
    public Boid(int BOX_WIDTH, ArrayList<Boid> flock) {
        this.flock = flock;
        this.BOX_WIDTH = BOX_WIDTH;
  } //Constructor
} //Boid
```

In order to flock, each `Boid` needs to have a position, velocity, and acceleration. Initially, the position can be anything within the SimBox and the velocity and acceleration may be random. To simulate `Boid` movement at some time step, the acceleration may be applied to the velocity (which should not exceed some maximum speed), and the velocity may be used to update the position.

```
float MAX_SPEED;
PVector pos, vel, acc;
class Boid {
    public Boid(int BOX_WIDTH, ArrayList<Boid> flock) {
        this.flock = flock;
        this.BOX_WIDTH = BOX_WIDTH;
        MAX_SPEED = 4;

        pos = PVector.random3D().mult(5);    //Controls how far Boids spawn from ea. other
        vel = PVector.random3D()
        acc = PVector.random3D();
  } //Constructor

  void flock() {
        vel.add(acc);           //Update velocity
        vel.limit(MAX_SPEED);
        acc.mult(0);            //Reset acceleration

        pos.add(vel);           //Update position
  } //flock
} //Boid
```

As aforementioned, global flocking behavior can be ascertained by applying three rules (forces) on each `Boid`'s acceleration, enumerated below:

1. Separation: Each Boid should move *away* from other Boids that are visible to it
2. Alignment: Each Boid should orient itself towards the direction other, visible Boids are moving
3. Cohesion: Each Boid should move *towards* the center of all the other Boids visible to it

Let's investigate each of these forces further.

2.3.2 Separation

For this explanation, let's focus on some `Boid`, b in a simulation of `Boids`, represented by the set B (thus $b \in B$). The separation force dictates that b should want to separate itself from other `Boids` that are visible to b. To determine which `Boids` are visible to b, we can define a visibility distance (`VISIBILITY`), and check every `Boid` $b' \in B, b \neq b'$ to see if the distance between $|b.\texttt{pos} - b'.\texttt{pos}| \leq \texttt{VISIBILITY}$. This is implemented as shown below:

```
private boolean isVis(Boid b) {
    float dist = pos.dist(b.pos);
    return dist <= VISIBILITY && dist > 0;
} //isVis
```

Now that we know which `Boids` are visible to b, we want to determine the proper separation force to apply to b. For each visible boid b', we can subtract b'`.pos` from b`.pos` to get a vector pointing *away* from b'. However, we want to separate b *more* from `Boids` that are quite close to b. To achieve this, we need to weight each direction vector inversely with the distance between b and b'. This is relatively straightforward to achieve by normalizing the direction vector and dividing it by the distance between b and b'. Finally, we find the average of all of these direction vectors for all visible b' `Boids`. Subtracting the resulting average direction vector by b's velocity (`vel`) gives us a final vector which points in the direction we should be pushing b's velocity to separate it from other nearby visible `Boids`.

2.3.3 Alignment

Let's investigate alignment by continuing the setup from separation. Consider each visible boid $b' \in V$ (the set of visible `Boids`). We want to point b's velocity towards the average velocity of each $b' \in V$. We can simply create an alignment vector (`ali`), then, in which we will accumulate the sum of `vel` for each $b' \in V$ and then divide `ali` by $|V|$. We can then subtract b`.vel` from `ali` as we did in separation, giving us our resulting force.

2.3.4 Cohesion

The logic for cohesion follows intuitively from alignment. Instead of pointing b's velocity towards the average b' velocity, however, we want to move b's velocity towards the center of all of $b' \in V$'s positions. This implementation is quite familiar – for every b', we want to sum their `pos` into a vector (`coh`) and then divide `coh` by $\|V\|$. Remember, though, this results in a vector with the average *position* – not a force! In order to create a force that pushes b towards this position, we need to subtract b`.pos` from `coh`, which then results in a vector pointing towards the direction we want b to move. We can then subtract b`.vel` from this vector.

2.3.5 Implementation

This can be implemented by generating a `PVector` for each force which can be applied to the `Boid`'s acceleration. The implementations for each force is shown in Listing 1.

```
    PVector sep = new PVector(0, 0, 0);
    int numVis = 0;

    for (Boid b : flock) {
        if (isVis(b)) {
          ++numVis;
          sep.add(
           PVector.sub(this.pos, b.pos)
           .normalize().div(getDist(b)));
        } //if
    } //for

    if (numVis > 0) {
      sep.div(numVis);
      sep.setMag(MAX_SPEED);
      sep.sub(vel);
      sep.limit(MAX_FORCE);
    } //if

    return sep;
```

```
    PVector coh = new PVector(0,0,0);
    int numVis = 0;

    for (Boid b : flock) {
      if (isVis(b)) {
        ++numVis;
        coh.add(b.pos);
      } //if
    } //for

    if (numVis > 0) {
      coh.div(numVis);
      PVector desired =
       PVector.sub(coh, this.pos);
      desired.setMag(MAX_SPEED);
      desired.sub(vel).limit(MAX_FORCE);
      return desired;
    }

    return coh;
```

```
    PVector ali = new PVector(0, 0, 0);
    int numVis = 0;

    for (Boid b : flock) {
      if (isVis(b)) {
        ++numVis;
        ali.add(b.vel);
      } //if
    } //for

    if (numVis > 0) {
      //find avg vel
      ali.div(numVis);
      //limit effect of force
      ali.setMag(MAX_SPEED);
      ali.sub(vel);
      ali.limit(MAX_FORCE);
    } //if

    return ali;
```

Listing 1: Implementations for `sep()`, `ali()`, and `coh()`

We may now update our `Boid`'s location using the `PVectors` of the aforedescribed forces in the previously written `flock()` method. Since it may also be desirable to change the weight assigned to each force (if, for example, we want Boids to separate from each other at all costs), we can also implement a simple weighting system which multiplies each force by a weight before being applied to a `Boid`'s acceleration vector, as shown below.

```
  void flock() {
        PVector sep = sep(), ali = ali(), coh = coh();  //Standard Boid forces

        sep.mult(SEP_WEIGHT); ali.mult(ALI_WEIGHT); coh.mult(COH_WEIGHT);
        acc.add(sep); acc.add(ali); acc.add(coh);

        vel.add(acc);           //Update velocity
        vel.limit(MAX_SPEED);
        acc.mult(0);            //Reset acceleration
        pos.add(vel);           //Update position
  } //flock
} //Boid
```

With that, we've succesfully imbued our `Boid`s with flocking behavior. In our main `draw()` loop, then, let's have each `Boid` we've created `flock()` as shown below.

```
void draw() {
  ...
  for (Boid b : flock) {
    b.flock();
  } //for
} //draw
```

2.4 Boid Rendering

Now that we have `Boids` that can move, we need a way of displaying them and their movement to our screen. There are plenty of options here ranging from simple and computationally inexpensive to intricate and cumbersome. We take a middle-of-the-line approach to balance aesthetics and performance.

Recall that each `Boid` has a position (`PVector pos`) and a velocity (`PVector vel`). Together, these tell us where a `Boid` is and where it is going, which is at the core of what we want to represent.

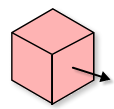


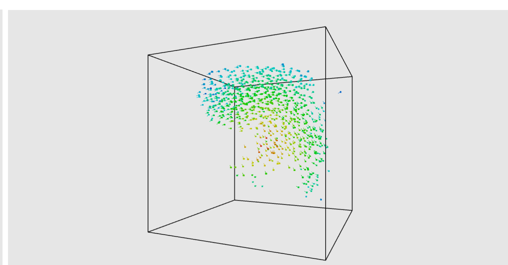
Figure 2: A simple `Boid` representation

To render this `Boid` in Processing, we will need a defined radius, a `box()` at the `Boid`'s `pos` and a `line()` from the `pos` pointing towards the `vel`. This is easy to achieve simply by moving our coordinate system to center at the `pos`, drawing a `box()`, then drawing a line from (0,0,0) to (`vel.x`, `vel.y`, `vel.z`) as implemented below.

```
public void display() {
    pushMatrix();
    translate(pos.x, pos.y, pos.z);
    PVector tip = PVector.mult(vel.normalize(null), 2*r);
    box(r);
    line(0,0,0,tip.x,tip.y,tip.z);
    popMatrix();
  } //display
```

With the rendering function finished, let's finally turn our attention back to the main `draw()` loop and have our `Boids` render themselves each frame.

```
...
void draw() {
  ...
  for (Boid b : flock) {
    b.flock();
    b.display();
  } //for
  ...
} //draw
```



2.5 Wall & Obstacle Avoidance

Now that we have our `Boids` have the ability to `flock()` and `display()`, we notice an interesting result – they fly out of our SimBox and eventually, out of our screen!

Thankfully, there are quite a few ways to keep our `Boids` contained. A simple approach might be to take turn every wall of our SimBox into a "portal" to the opposite side of the SimBox. If a `Boid` flies out the right side of the SimBox, they'll reappear on the left! Similarly, if they fly off the top, they'll reappear on the bottom. Thankfully, we were careful to pass in our `BOX_WIDTH` earlier, so this "bounding" is trivial to implement:

```java
private void bound() {
   pos.x = pos.x <= -BOX_WIDTH/2 ?  BOX_WIDTH/2 - 10: pos.x >= BOX_WIDTH/2 ? -BOX_WIDTH/2 + 10 : pos.x;
   pos.y = pos.y <= -BOX_WIDTH/2 ?  BOX_WIDTH/2 - 10: pos.y >= BOX_WIDTH/2 ? -BOX_WIDTH/2 + 10 : pos.y;
   pos.z = pos.z <= -BOX_WIDTH/2 ?  BOX_WIDTH/2 - 10: pos.z >= BOX_WIDTH/2 ? -BOX_WIDTH/2 + 10 : pos.z;
 } //bound
```

Unfortunately, this introduces a new problem! Now, when a small flock of `Boids` flies through a "portal", they can no longer see their neighbors who have reappeared on the opposite side because the `PVector.dist()` function isn't accounting for our "portal" mechanics. Though it is possible to fix this problem, it may be more elegant to discourage `Boids` from moving through walls altogether. To do this, we want to imbue our `Boids` with the ability to avoid certain areas whilst still `flock()`ing.

Notice that we've already created a function similar to this – the `sep()` function which separates `Boids` that are too close to one another *avoids* other `Boids` based on their position. Drawing inspiration from `sep()`, then, for each wall, our `Boid` should (1) create a `steer` vector pointing opposite the wall, (2) weight the `steer` vector by how close the wall is to the current `pos`, and add that to a final avoidance (`avo`) vector. Once this has been calculated for each wall, we can repeat the logic from `sep` and divide `avo` by 6 (the number of walls), limit it to the `MAX_FORCE` allowed, and return it. This is implemented below.

```java
private PVector wallAvoid() {
   PVector avo = new PVector(0, 0, 0);
   PVector walls [] = {new PVector(-BOX_WIDTH/2, pos.y, pos.z),
                       new PVector(BOX_WIDTH/2, pos.y, pos.z),
                       new PVector(pos.x, -BOX_WIDTH/2, pos.z),
                       new PVector(pos.x, BOX_WIDTH/2, pos.z),
                       new PVector(pos.x, pos.y, -BOX_WIDTH/2),
                       new PVector(pos.x, pos.y, BOX_WIDTH/2)};

   for (PVector wall : walls) {
     PVector steer = new PVector();
     steer.set(PVector.sub(pos,wall).normalize());
     steer.mult(1/PVector.dist(pos,wall));  //Weight by proximity
     avo.add(steer);
   } //for

   avo.div(6).limit(MAX_FORCE);
   return avo;
 } //wallAvoid
```

```java
public void flock() {
    PVector sep = sep(), ali = ali(), coh = coh();
    PVector avo = wallAvoid();

    //Apply weights to each force
    sep.mult(SEP_WEIGHT);
    ali.mult(ALI_WEIGHT);
    coh.mult(COH_WEIGHT);
    avo.mult(WALL_WEIGHT);

    acc.add(sep);
    acc.add(ali);
    acc.add(coh);
    acc.add(avo);

    ...
    pos.add(vel); //Update position
    bound();      //Handle any OOB
  } //flock
```

With wall-avoidance in place, `Boids` still sometimes go out-of-bounds, reappearing on the other side of the SimBox, but this happens far less frequently.

Interestingly, this avoidance function can be applied to more than just walls. We can modify minimal logic from `wallAvoid()` to avoid obstacles more broadly. To test how this might work, let's create an `Obstacle` class. Since `wallAvoid()` is based solely off the wall's position, our `Obstacle` only needs to have a position and a method of displaying itself. This can be implemented as follows:

```
public class Obstacle {
  PVector pos;
  float r; //radius

  public Obstacle(PVector pos, float r) {
    this.pos = pos;
    this.r = r;
  } //Constructor

  public void display() {
    pushMatrix();
    translate(pos.x, pos.y, pos.z);
    sphere(r/2);
    popMatrix();
  } //display
} //Obstacle
```

```
private PVector avoid(Obstacle o) {
    PVector avo = new PVector();

    //If out of eyesight
    if (pos.dist(o.pos) > o.r + VISIBILITY) {
        return avo;
    } //if

    //Find vector pointing in opp direction
    avo.set(PVector.sub(pos,o.pos));
    //Weight by proximity
    avo.mult(1/PVector.dist(pos,o.pos));
    avo.limit(MAX_FORCE);

    return avo;
  } //avoid
```

Now, we repeat the process of adding `Obstacle`s into the `BoidRunner` as we did with `Boid`s, and we add the `avo` force to `acc` just as we did with the other forces.

```
final int NUM_OBSTACLES;
ArrayList<Obstacle> obstacles;
...
void setup() {
  fullScreen(P3D);
  noFill();
  stroke(0);
  ...
  //Random Obstacle Setup
  obstacles = new ArrayList<Obstacle> ();
  for (int i = 0; i < NUM_OBSTACLES; i++) {
    float r = random(50, 90);
    obstacles.add(new Obstacle(new PVector(...)));
  } //for
} //setup

void draw() {
...
  for (Obstacle o : obstacles) {
    o.display();
  } //for

  for (Boid b : boids) {
    b.flock();
    b.display();
  } //for
...
} //draw
```

```
 public void flock() {
    PVector sep = sep(), ali = ali(), coh = coh();
    PVector avo = wallAvoid();

    ...
    //Apply weights to each force
    sep.mult(SEP_WEIGHT);
    ali.mult(ALI_WEIGHT);
    coh.mult(COH_WEIGHT);
    avo.mult(WALL_WEIGHT);

    ...
    //Generate acceleration force
    for (Obstacle o : obstacles) {
      acc.add(avoid(o).mult(OBS_WEIGHT));
    } //for
    acc.add(sep);
    acc.add(ali);
    acc.add(coh);
    acc.add(avo);

    vel.add(acc); //Update velocity
    vel.limit(MAX_SPEED);
    acc.mult(0);  //Reset acceleration

    pos.add(vel); //Update position
    bound();      //Handle any OOB
  } //flock
```

Altogether, this leaves us with our penultimate results in Figure 3.

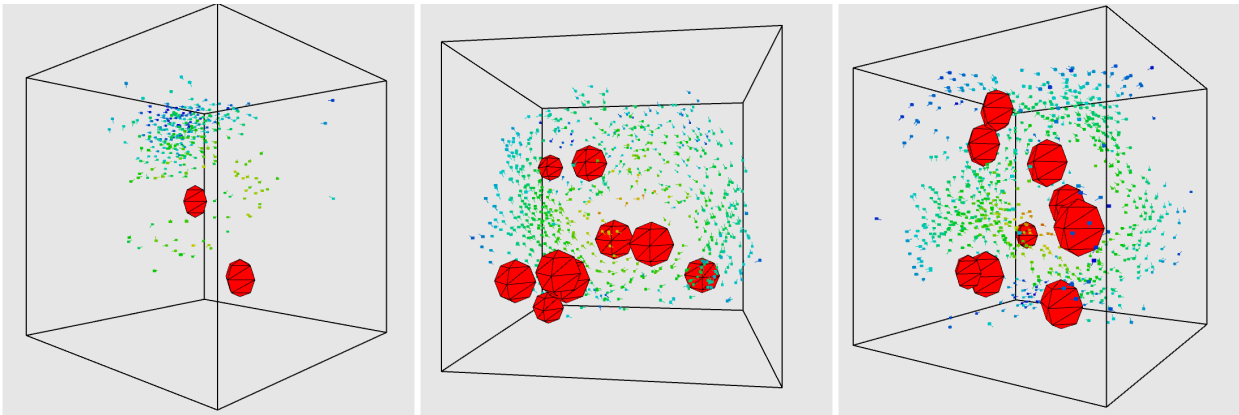


Figure 3: `Boids` avoiding different `Obstacles`

2.6 GUI Development

While we now have a completely functioning `Boids` simulation, it would be nice to increase the amount of user interactivity present. Most notably, it would be nice to let users create `flock()`ing behavior of their own, changing the weights of different forces to create `Boids` that behave in unique ways. Specifically, we'd like to let user's control:

- Separation weight
- Alignment weight
- Cohesion weight
- Wall avoidance weight
- Obstacle avoidance weight
- Visibility
- Number of Boids

To do this, we use the `ControlP5` library[3]. The library requires minimal setup to the `BoidRunner` as follows:

```
import controlP5.*;

/* ============ G L O B A L S ============ */
final int BOX_WIDTH = 450;
...
ControlP5 cp5;

void setup() {
  ...
  cp5 = new ControlP5(this);
```

Adding in sliders is relatively straightforward for each parameter as well. To do this, we create a simple helper function. Every frame, we want to poll each of our sliders to see if any of their values have changed. If they have, we want to update our weights to the value on the slider. This is implemented as shown below.

[3]https://www.sojamo.de/libraries/controlP5/

```
void initGUI() {
  cp5.addSlider("SEPARATION WEIGHT")
       .setFont(font)
       .setPosition(40, 40)
       .setSize(100, 20)
       .setRange(0, 10)
       .setValue(3.5)
       .setColorCaptionLabel(color(255));
  ...
  //repeated for other params
} //initGUI
```

```
void draw() {
    ...
    int numDesired = cp5.getController("BOIDS".getValue());
    if (NUM_BOIDS != numDesired) {
        NUM_BOIDS = numDesired;
        boids = new ArrayList<Boid> ();
        for (int i = 0; i < NUM_BOIDS; i++) {
          boids.add(new Boid(BOX_WIDTH, boids, obstacles));
        } //for
    } //if
} //draw
```

This leaves us with the final simulation window shown in Figure 4.

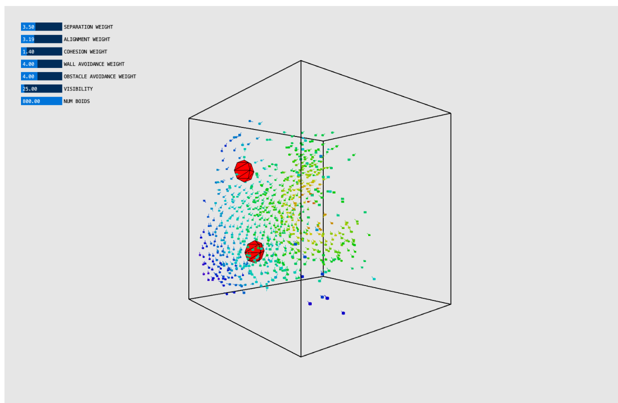


Figure 4: Final Sim with GUI

3 Conclusion & Discussion

Here, we've developed a 3D Boid simulation that builds atop the canonical model originally presented by Reynolds. Notably, we add significant amounts of user-interactivity in the form of scene rotation implemented via the `Arcball` and `Quat` classes and through real-time `Boid` parameter tuning via the `ControlP5` library. We've additionally added features to the `Boid`s which supplement their basic `flock()`ing functionality to include `Obstacle` and wall avoidance. Finally, we've also made some aesthetic changes to `Boid` representation such that they are dynamically colored based on their distance from the center (which generally serves as a good proxy for coloring by flock).

This is a useful exercise for intermediate users of the Processing software, as it incorporates the use of different classes and Object Oriented Programming (OOP) techniques while also engaging critically with ideas undergirding physics animations such as the interaction of forces. The project also engages with computer graphics content by (1) creating a `SimBox` that is resistant to roll and uses standard arcballs and quaternions to handle rotation, (2) manipulating various coordinate systems to handle the rendering of the SimBox, each `Boid`, and each `Obstacle`, and (3) developing a simple GUI.

Future improvements to this simulation include design and computation optimizations. Most notably, we could improve the representation of a `Boid`. Instead of rendering it as a simple box with a direction, we could use a more detailed model by loading in `.obj` files which are supported by Processing via `PShape`s (though this would incur a computational overhead). Additionally, we could apply more complicated shaders, introducing interactions with light sources or reflective materials more than the basic ambient light currently present.

Computationally, our current implementation checks every pair of `Boid`s to see if they're visible to one another and calculating the relevant forces between them, if applicable. Though this works quickly enough to not cause frame lag with a low number of `Boid`s, the computation required for this approach scales quadratically and will quickly slow down the simulation. Future work might seek to implement spatial hashing to optimize the `Boid` update step, effectively breaking the SimBox into a grid-like structure. This allows us to check only *nearby* `Boid`s for force interactions, potentially reducing the number of computations required for `flock()`ing by orders of magnitude.