Game developers are, in greater and greater numbers, turning to openFrameworks' creative coding toolkit to develop their games. Unlike platforms like Unity, GameMaker, and Construct2, oF was not specifically developed for game makers. However, oF's ability to port to mobile, manipulate video, utilize camera input, support generative graphics, and hook in with devices like the Arduino and Kinect (among other features) makes it a very attractive option for developers who want to be able to rapidly produce compelling, unique games.

[image and caption of Eliss]

[image and caption of worm run]

[image and caption of spell tower]

[image and caption of drop blocs]

[image and caption of scream em up? or another more experimental oF game?]

In this chapter, we'll learn about game development in openFrameworks. We'll cover what goes into making a game, as well as how to code a simple space shooter. Finally, we'll put an experimental oF twist on our game by implementing OSC functionality, which will allow you to alter the difficulty of the game live—while a player is playing it.

Ready? Let's go!

There are as many ways to make games as there are game developers. However, many developers follow an **iterative process**: that is, adding a single component, testing it, adding an additional component, testing it again, and so on. Regardless of the platform, this method allows game developers to quickly figure out what parts of the initial idea are worth keeping and rapidly test additions they think might be interesting--without having to risk wasting time on building out a complete game that, in retrospect, isn't compelling.

This iterative process can be done digitally or physically. **Paper prototyping** is the process of testing mechanics and interactions with paper models and analogs. Although these paper prototypes don't necessarily look like the final game, they can be mocked up quickly and thrown away cheaply, allowing developers to experiment with core mechanics more rapidly than they could with code. For example, a puzzle game's board and pieces can likely be mocked up with paper and dice more quickly than it can be implemented in even a basic mobile app. When a developer makes a **digital prototype**, or one made with code, they will similarly typically start by refining game mechanics, keeping assets rough until they get closer to the end. Finally, developers enter the long process of **tuning** their game, tweaking various parameters about the game until it feels just right.

We're going to use openFrameworks to play with the final step of this process. In the game we're making, we're not going to settle on one set of parameters that stay static from game to game. We're going to use openFrameworks' OSC library to allow us to communicate wirelessly from another device (e.g. a smartphone or table) so we can tune those parameters live, giving our players experiences tailored just for them.

So what is OSC anyway? OSC, or Open Sound Control, came about as an advancement to MIDI, so let's talk about MIDI first. MIDI is a data protocol that sends and receives information between devices, typically electronic musical instruments. MIDI is what allowed things like keyboards and drum machines to fire in sync. If you've heard pop music, you've heard MIDI in action.

MIDI has data channels, on which you can send or receive single messages, or events. Programmers could associate these MIDI events with actions that their electronic instruments could take. For example, you could set up your keyboard to send data on channel 1, and receive data on MIDI channel 2. More specifically, you could program a specific key (say, the 'a' key) to send out a MIDI event on channel 1. If your drum machine is set up to receive on channel 1, it will receive that message and perform the appropriate action (e.g. playing). A pretty cool system, but one that was limited by its pre-defined and discrete message types.

As time advanced, so did computers and the speed of data transfers, leading us to OSC. OSC was designed to allow for more expressive performance data, with different, flexible kinds of messages sent over networks. OSC is a thin layer on top of the UDP protocol, and allows users to send information over networks just by specifying the network address and the incoming and outgoing ports. (UDP is used frequently in games, and it is possible to use both of these protocols at the same time in the same code base with no issues.)

OSC messages consist of the following:

1. **An address pattern.** This is a hierarchical name space, and looks a bit like a Unix filesystem or URL (e.g. “/Address1”). These patterns can effectively be anything you want (e.g. “/EnemySpeed”)--think of them as names for what you send.
2. **A Type tag string.** This simply represents the kind of data being sent (e.g. int, string).
3. **Arguments.** The actual value that is being transmitted (e.g. 6, “Hello world”, etc.).

There are plenty of inexpensive apps for smartphones and tablets that provide customizable GUIs (complete with buttons, sliders, etc.) for sending different kinds of MIDI messages. Download one of the following ([name, name, name]) so we have something to send our messages with.

With this basic understanding in mind, let's start making our game!

OpenFrameworks handles OSC as an included addon, so our first step will be to run the project generator and create a project with the OSC addon. (If you haven't had a chance to read about addons, now would be a good time to jump over to [here] and do just that!) Launch the project generator, then, in the main menu, click the word “Addons.” A popup will appear. Select ofxOsc and then click back. Now, next to the word Addons, you should see ofxOsc. Press “generate”. When it completes the project creation process, close the generator and open up the project in either Visual Studio[[1]](#footnote-1) or Xcode. The project will be set up in your myApps folder. Open it now.

Here's what our game will have:

* A player, who has an on-screen position, a movement speed, and an image to represent it
* Some enemies, who have an on-screen position, a movement speed (with the horizontal value based on a sine wave), an image to represent them, and an interval to keep track of when they can shoot next
* A level controller, which has an interval to keep track of when an enemy should be spawned next
* Bullets (for the player and the enemies), which have an on-screen position, images to represent them, a way to keep track of where they come from (player or enemy), and a speed
* Extra lives, which have an on-screen position, an image to represent them, and a speed

With all that written out, let’s use OSC to affect the following:

* The horizontal movement of our enemies--whether they move in a more exaggerated sin wave, or whether they move in more of a straight line
* The frequency with which our enemies shoot
* The frequency with which our level controller spawn enemies
* Whether a life bonus is on screen or not

These three parameters will allow the developer to, second-by-second, tailor the difficulty of the game to the individual playing it.

Let’s start with our testApp. There are a few things we definitely know we’ll want classes for, so make corresponding .h and .cpp files for Player, Bullet, Life, Enemy, and LevelController.



Remember to #include “ofMain.h” in each of those classes, and to include the .h file of each of those classes in testApp.h.

First let’s create the basic structure of our game. Games typically have at least three parts: a start screen, the game itself, and an end screen. We need to keep track of which section of the game we’re in, which we’ll do using a variable called a game state. In this example, our game state variable is a string, and the three parts of our game are “start”, “game”, and “end”. Let’s add a score and a player at this point as well.

string game\_state;

int score;

Player player\_1;

We’ll then divide up testApp’s update() and draw() loops between those game states:

//--------------------------------------------------------------

void testApp::update(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

   } else if (game\_state == “end”) {

   }

}

//--------------------------------------------------------------

void testApp::draw(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

   } else if (game\_state == “end”) {

   }

}

Let’s set the initial value of game\_state to “start” right when the app begins.

//--------------------------------------------------------------

void testApp::setup(){

   game\_state = "start";

  score = 0;

}

Finally, let’s make sure that we can move forward from the start screen. In this example, when the player is on the start screen and releases the space key, they’ll be taken to the game.

//--------------------------------------------------------------

void testApp::keyReleased(int key){

   if (game\_state == "start") {

       game\_state = "game";

   } else if (game\_state == "game") {

      // blank for now

   }

Great! Let’s move onto our player. Our player’s class looks like this:

class Player {

public:

   ofPoint pos;

   float width, height, speed;

   int lives;

   bool is\_left\_pressed, is\_right\_pressed, is\_down\_pressed, is\_up\_pressed;

   void setup(ofImage \* \_img);

   void update();

   void draw();

   void shoot();

   void calculate\_movement();

   bool check\_can\_shoot();

   ofImage \* img;

};

Taking this one step at a time:

* Our player’s position will be stored in an ofPoint called pos. ofPoints are handy datatypes that contain x and y values, letting us access our player’s position through pos.x and pos.y.
* Our player will have width, height, and speed variables (which we’ll use for collision detection and movement, respectively).
* Our player will have an integer number of lives (since it wouldn’t make any sense for them to have 4.33333333333 lives)
* Our player will keep track of what movement keys are currently pressed in separate booleans
* Our player will have setup, update, draw, shoot, and calculate\_movement methods.
* Finally, our player will have a pointer to the image we’re using for the player.

You may be wondering why we’re using all these booleans--why not just check and see which keys are pressed?

The problem is that, in openFrameworks, keyPressed() does not return all the keys currently being pressed--just the last key that was pressed. That means that if the player presses up and left (intending to move diagonally), openFrameworks will only report one of the keys being pressed. You can try printing out the result of keyPressed to see this in action.

What we’ll do to avoid this is instead base the player’s movement on the booleans we wrote earlier. If the player presses a certain key, that boolean will be true; if they release that key, that boolean will be false. That way, if the player presses up and left, we’ll report up and left as being true until those keys are released.

Here’s what our new keyPressed() and keyReleased() functions look like:

//--------------------------------------------------------------

void testApp::keyPressed(int key){

   if (game\_state == "game") {

       if (key == OF\_KEY\_LEFT) {

           player\_1.is\_left\_pressed = true;

       }

       if (key == OF\_KEY\_RIGHT) {

           player\_1.is\_right\_pressed = true;

       }

       if (key == OF\_KEY\_UP) {

           player\_1.is\_up\_pressed = true;

       }

       if (key == OF\_KEY\_DOWN) {

           player\_1.is\_down\_pressed = true;

       }

   }

}

//--------------------------------------------------------------

void testApp::keyReleased(int key){

   if (game\_state == "start") {

       game\_state = "game";

   } else if (game\_state == "game") {

       if (key == OF\_KEY\_LEFT) {

           player\_1.is\_left\_pressed = false;

       }

       if (key == OF\_KEY\_RIGHT) {

           player\_1.is\_right\_pressed = false;

       }

       if (key == OF\_KEY\_UP) {

           player\_1.is\_up\_pressed = false;

       }

       if (key == OF\_KEY\_DOWN) {

           player\_1.is\_down\_pressed = false;

       }

   }

}

Add ofImage player\_image to testApp.h, then load the player’s image and instantiate the player in testApp’s setup():

void testApp::setup(){

   game\_state = "start";

   player\_image.loadImage("player.png");

   player\_1.setup(&player\_image);

}

Finally, update and draw your player in the appropriate part of testApp::update() and testApp::draw():

//--------------------------------------------------------------

void testApp::update(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       player\_1.update();

   }

}

//--------------------------------------------------------------

void testApp::draw(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       player\_1.draw();

   } else if (game\_state == "end") {

   }

}

You should have a player who moves around on-screen. Sweet!

Let’s make our bullets next. In order to have a variable number of bullets on screen at a time, we need to add a vector<Bullet> bullets to testApp.h. Let’s also create a void update\_bullets(); function, which will update our vector of bullets (and, shortly, trigger the check for bullet collisions). We also want our player and enemy bullets to look different, so we’ll add ofImage enemy\_bullet\_image; and ofImage player\_bullet\_image; to our testApp.h file.

Our bullet class will look a lot like the player class, having a position, speed, width, pointer to an image, and various functions. The big difference is that the bullets will keep track of who they came from (since that will affect who they can hurt and which direction they move).

class Bullet {

public:

   ofPoint pos;

   float speed;

   float width;

   bool from\_player;

   void setup(bool f\_p, ofPoint p, float s, ofImage \* bullet\_image);

   void update();

   void draw();

   ofImage \* img;

};

Our Bullet.cpp will look like this:

void Bullet::setup(bool f\_p, ofPoint p, float s, ofImage \* bullet\_image) {

   from\_player = f\_p;

   pos = p;

   speed = s + 3;

   img = bullet\_image;

   width = img->width;

}

void Bullet::update() {

   if (from\_player) {

       pos.y -= speed;

   } else {

       pos.y += speed;

   }

}

void Bullet::draw() {

   img->draw(pos.x - width/2, pos.y - width/2);

}

Again, this is much like the code for the player. The two differences are:

* We keep track of where the bullet comes from, and alter the code based on that variable (meaning we can keep all the bullets in the same vector)
* When instantiating a bullet, we check to see the position of the shooter, as well as the shooter’s current speed (so it will always move faster than the thing that shot it)

Now that our bullet class is implemented, we can go back to testApp::setup() and add in enemy\_bullet\_image.loadImage("enemy\_bullet.png"); and player\_bullet\_image.loadImage("player\_bullet.png"); right underneath where we loaded in our player\_image.

For now, our update\_bullets function will call the update() function in each bullet, and will also get rid of bullets that have flown offscreen in either direction.

//--------------------------------------------------------------

void testApp::update\_bullets() {

   for (int i = 0; i < bullets.size(); i++) {

       bullets[i].update();

       if (bullets[i].pos.y - bullets[i].width/2 < 0 || bullets[i].pos.y + bullets[i].width/2 > ofGetHeight()) {

           bullets.erase(bullets.begin()+i);

       }

   }

   // we’ll call a collision check function here shortly

}

Our testApp::update() and testApp::draw() will now look like this:

//--------------------------------------------------------------

void testApp::update(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       player\_1.update();

       update\_bullets();

   }

}

//--------------------------------------------------------------

void testApp::draw(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       ofBackground(0,0,0);

       player\_1.draw();

       for (int i = 0; i < bullets.size(); i++) {

           bullets[i].draw();

       }

   } else if (game\_state == "end") {

   }

}

Finally, let’s add an if-statement to our keyPressed() so that when we press the spacebar during the game, we spawn a player bullet:

//--------------------------------------------------------------

void testApp::keyPressed(int key){

   if (game\_state == "game") {

       if (key == OF\_KEY\_LEFT) {

           player\_1.is\_left\_pressed = true;

       }

       if (key == OF\_KEY\_RIGHT) {

           player\_1.is\_right\_pressed = true;

       }

       if (key == OF\_KEY\_UP) {

           player\_1.is\_up\_pressed = true;

       }

       if (key == OF\_KEY\_DOWN) {

           player\_1.is\_down\_pressed = true;

       }

       if (key == ' ') {

           Bullet b;

           b.setup(true, player\_1.pos, player\_1.speed, &player\_bullet\_image);

           bullets.push\_back(b);

       }

   }

}

Remember, the first parameter in the bullet’s setup is whether it comes from the player (which, in this case, is always true). Run your app and fly around shooting for a bit to see how it feels.

Let’s move on to our enemy. This process should be familiar by now. Add an ofImage enemy\_image; and a vector<Enemy> enemies; to testApp.h. Additionally, add float max\_enemy\_amplitude; and float max\_enemy\_shoot\_interval; to testApp.h--these are two of the enemy parameters we’ll affect with OSC.

Your enemy class will look like this:

class Enemy {

public:

   ofPoint pos;

   float speed;

   float amplitude;

   float width;

   float start\_shoot;

   float shoot\_interval;

   void setup(float m\_e\_a, float m\_e\_s\_i, ofImage \* enemy\_image);

   void update();

   void draw();

   bool time\_to\_shoot();

   ofImage \* img;

};

Our enemy’s horizontal movement will be shaped by the values fed to a sine wave (which we’ll see in a moment). We’ll keep track of our amplitude variable (so different enemies can have different amplitudes). We’ll also want to keep track of whether enough time has passed for this enemy to shoot again, necessitating the start\_shoot and shoot\_interval variables. Both of these variables will actually be set in our setup() function, with m\_e\_a being the max enemy amplitude, and m\_e\_s\_i being the max enemy shooting interval. Finally, we’ll have a boolean function that will tell us whether the enemy can shoot this frame or not.

Our enemy class will look like this:

void Enemy::setup(float m\_e\_a, float m\_e\_s\_i, ofImage \* enemy\_image) {

   pos.x = ofRandom(ofGetWidth());

   pos.y = 0;

   img = enemy\_image;

   width = img->width;

   speed = ofRandom(2, 7);

   amplitude = ofRandom(m\_e\_a);

   shoot\_interval = ofRandom(0.5, m\_e\_s\_i);

   start\_shoot = ofGetElapsedTimef();

}

void Enemy::update() {

   pos.y += speed;

   pos.x += amplitude \* sin(ofGetElapsedTimef());

}

void Enemy::draw() {

   img->draw(pos.x - width/2, pos.y - width/2);

}

bool Enemy::time\_to\_shoot() {

   if (ofGetElapsedTimef() - start\_shoot > shoot\_interval) {

       start\_shoot = ofGetElapsedTimef();

       return true;

   }

   return false;

}

In update, we’re using the current elapsed time in frames to give us a constantly increasing number to feed to the sine function, which in turn returns a value between -1 and 1. We multiply it by the amplitude of the wave, making this curve more or less exaggerated.

In time\_to\_shoot, we check to see whether the difference between the current time and the time this enemy last shot is greater than the enemy’s shooting interval. If it is, we set start\_shoot to the current time, and return true. If not, we return false.

Let’s integrate our enemies into the rest of our testApp.cpp:

//--------------------------------------------------------------

void testApp::setup(){

   game\_state = "start";

   max\_enemy\_amplitude = 3.0;

   max\_enemy\_shoot\_interval = 1.5;

   enemy\_image.loadImage("enemy0.png");

   player\_image.loadImage("player.png");

   enemy\_bullet\_image.loadImage("enemy\_bullet.png");

   player\_bullet\_image.loadImage("player\_bullet.png");

   player\_1.setup(&player\_image);

}

//--------------------------------------------------------------

void testApp::update(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       player\_1.update();

       update\_bullets();

       for (int i = 0; i < enemies.size(); i++) {

           enemies[i].update();

           if (enemies[i].time\_to\_shoot()) {

               Bullet b;

               b.setup(false, enemies[i].pos, enemies[i].speed, &enemy\_bullet\_image);

               bullets.push\_back(b);

           }

       }

   } else if (game\_state =="draw") {

   }

}

//--------------------------------------------------------------

void testApp::draw(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       ofBackground(0,0,0);

       player\_1.draw();

       for (int i = 0; i < enemies.size(); i++) {

           enemies[i].draw();

       }

       for (int i = 0; i < bullets.size(); i++) {

           bullets[i].draw();

       }

   } else if (game\_state == "end") {

   }

}

Let’s implement our bullet collision checks. Add a void check\_bullet\_collisions(); to your testApp.h, then write the following function:

//--------------------------------------------------------------

void testApp::check\_bullet\_collisions() {

   for (int i = 0; i < bullets.size(); i++) {

       if (bullets[i].from\_player) {

           for (int e = enemies.size()-1; e >= 0; e--) {

               if (ofDist(bullets[i].pos.x, bullets[i].pos.y, enemies[e].pos.x, enemies[e].pos.y) < (enemies[e].width + bullets[i].width)/2) {

                   enemies.erase(enemies.begin()+e);

                   bullets.erase(bullets.begin()+i);

                   score+=10;

               }

           }

       } else {

           if (ofDist(bullets[i].pos.x, bullets[i].pos.y, player\_1.pos.x, player\_1.pos.y) < (bullets[i].width+player\_1.width)/2) {

               bullets.erase(bullets.begin()+i);

               player\_1.lives--;

               if (player\_1.lives <= 0) {

                   game\_state = "end";

               }

           }

       }

   }

}

This code is a bit nested, but actually pretty simple. First, it goes through each bullet in the vector and checks to see whether it’s from the player. If it’s from the player, it starts a for-loop for all the enemies, so we can compare the player bullet position against all the enemy positions. We use ofDist() to see whether the distance between a given bullet and a given enemy is less than the sum of their radii--if it is, they’re overlapping.

If a bullet is not from the player, the function does a distance calculation against the player, to see whether a given enemy bullet and the player are close enough to count it as a hit. If there is a hit, we subtract a player’s life and erase that bullet. If the player has less than or equal to 0 lives, we change the game state to the end.

Don’t forget to call check\_bullet\_collisions() as part of update\_bullets():

//--------------------------------------------------------------

void testApp::update\_bullets() {

   for (int i = 0; i < bullets.size(); i++) {

       bullets[i].update();

       if (bullets[i].pos.y - bullets[i].width/2 < 0 || bullets[i].pos.y + bullets[i].width/2 > ofGetHeight()) {

           bullets.erase(bullets.begin()+i);

       }

   }

   check\_bullet\_collisions();

}

Great! Except… we don’t have any enemies yet! Definitely an oversight. This is where our level controller comes in. Add LevelController level\_controller; to your testApp.h.

Our level controller class is super-simple:

class LevelController {

public:

   float start\_time;

   float interval\_time;

   void setup(float e);

   bool should\_spawn();

};

As you might guess, all it’ll really do is keep track of whether it’s time to spawn another enemy yet.

Inside our LevelController.cpp:

void LevelController::setup(float s) {

   start\_time = s;

   interval\_time = 500;

}

bool LevelController::should\_spawn() {

   if (ofGetElapsedTimeMillis() - start\_time > interval\_time) {

       start\_time = ofGetElapsedTimeMillis();

       return true;

   }

   return false;

}

When we set up our level controller, we’ll give it a starting time. It’ll use this time as a baseline for the first enemy spawn. The should\_spawn code should look familiar from the enemy bullet section.

We’ll wait to set up our level controller until the game actually starts--namely, when the game state changes from “start” to “game”.

void testApp::keyReleased(int key){

   if (game\_state == "start") {

       game\_state = "game";

       level\_controller.setup(ofGetElapsedTimeMillis());

   }

  ...

}

Next we’ll integrate it into our testApp::update():

//--------------------------------------------------------------

void testApp::update(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       player\_1.update();

       update\_bullets();

       for (int i = 0; i < enemies.size(); i++) {

           enemies[i].update();

           if (enemies[i].time\_to\_shoot()) {

               Bullet b;

               b.setup(false, enemies[i].pos, enemies[i].speed, &enemy\_bullet\_image);

               bullets.push\_back(b);

           }

       }

       if (level\_controller.should\_spawn() == true) {

           Enemy e;

           e.setup(max\_enemy\_amplitude, max\_enemy\_shoot\_interval, &enemy\_image);

           enemies.push\_back(e);

       }

   }

}

Awesome! We’re close to done!

Before we finish, let’s add in our last OSC feature: the ability to throw in bonus lives on the fly. Add vector<Life> bonuses; and ofImage life\_image; to your testApp.h. To keep our code modular, let’s also add void update\_bonuses(); in the same place. Don’t forget to life\_image.loadImage("life\_image.png"); in testApp::setup().

Life.h should look like this:

class Life {

public:

   ofPoint pos;

   float speed;

   float width;

   ofImage \* img;

   void setup(ofImage \* \_img);

   void update();

   void draw();

};

And it’ll function like this--a lot like the bullet:

void Life::setup(ofImage \* \_img) {

   img = \_img;

   width = img->width;

   speed = 5;

   pos.x = ofRandom(ofGetWidth());

   pos.y = -img->width/2;

}

void Life::update() {

   pos.y += speed;

}

void Life::draw() {

   img->draw(pos.x - img->width/2, pos.y - img->width/2);

}

Our update\_bonuses() function works a lot like the bullet collision function:

//--------------------------------------------------------------

void testApp::update\_bonuses() {

   for (int i = bonuses.size()-1; i > 0; i--) {

       bonuses[i].update();

       if (ofDist(player\_1.pos.x, player\_1.pos.y, bonuses[i].pos.x, bonuses[i].pos.y) < (player\_1.width + bonuses[i].width)/2) {

           player\_1.lives++;

           bonuses.erase(bonuses.begin() + i);

       }

       if (bonuses[i].pos.y + bonuses[i].width/2 > ofGetHeight()) {

           bonuses.erase(bonuses.begin() + i);

       }

   }

}

All that’s left for our lives functionality is to alter testApp::update() and testApp::draw().

//--------------------------------------------------------------

void testApp::update(){

   if (game\_state == "start") {

   } else if (game\_state == "game") {

       player\_1.update();

       update\_bullets();

       update\_bonuses();

       for (int i = 0; i < enemies.size(); i++) {

           enemies[i].update();

           if (enemies[i].time\_to\_shoot()) {

               Bullet b;

               b.setup(false, enemies[i].pos, enemies[i].speed, &enemy\_bullet\_image);

               bullets.push\_back(b);

           }

       }

       if (level\_controller.should\_spawn() == true) {

           Enemy e;

           e.setup(max\_enemy\_amplitude, max\_enemy\_shoot\_interval, &enemy\_image);

           enemies.push\_back(e);

       }

   }

}

//--------------------------------------------------------------

void testApp::draw(){

   if (game\_state == "start") {

       start\_screen.draw(0,0);

   } else if (game\_state == "game") {

       ofBackground(0,0,0);

       player\_1.draw();

       draw\_lives();

       for (int i = 0; i < enemies.size(); i++) {

           enemies[i].draw();

       }

       for (int i = 0; i < bullets.size(); i++) {

           bullets[i].draw();

       }

       for (int i = 0; i < bonuses.size(); i++) {

           bonuses[i].draw();

       }

   } else if (game\_state == "end") {

   }

}

Finally, we’ve been a bit stingy with visual feedback, so let’s add in a start screen, a score, a visual representation of the lives left, and an end screen. Add ofImage start\_screen; , ofImage end\_screen; , void draw\_lives(); , and void draw\_score(); to testApp.h.

Change testApp::setup() to load in those assets:

//--------------------------------------------------------------

void testApp::setup(){

   ...

   player\_1.setup(&player\_image);

   start\_screen.loadImage("start\_screen.png");

   end\_screen.loadImage("end\_screen.png");

   score\_font.loadFont("Gota\_Light.otf", 48);

}

Draw them in the appropriate game states using start\_screen.draw(0, 0); and end\_screen.draw(0, 0);.

Add in the last two functions:

//--------------------------------------------------------------

void testApp::draw\_lives() {

   for (int i = 0; i < player\_1.lives; i++) {

       player\_image.draw(ofGetWidth() - (i \* player\_image.width) - 100, 30);

   }

}

//--------------------------------------------------------------

void testApp::draw\_score() {

   if (game\_state == "game") {

       score\_font.drawString(ofToString(score), 30, 72);

   } else if (game\_state == "end") {

       float w = score\_font.stringWidth(ofToString(score));

       score\_font.drawString(ofToString(score), ofGetWidth()/2 - w/2, ofGetHeight()/2 + 100);

   }

}

By using stringWidth(), we can calculate the width of a string--handy for centering it.

All that’s left after that is to call draw\_score(); and draw\_lives(); during the testApp::draw()’s game state, and to call draw\_score(); during the end state.

Congrats--you made a game!

Now let’s add in the OSC functionality. We are going to set our application up to receive messages from our iPad and then make changes in real-time while our game is running to test an array of possible play scenarios. Then we are going to give ourselves some visual feedback so we know what our current settings are. [NOTE THIS MIGHT BE THE MOMENT FOR A SETTINGS FILE] As mentioned before, this can trump going into your application and making manual changes because you skip the need to recompile your game. This is something you can even do from an iphone while someone else plays your game on an iPad, completely saving the need for rebuilding out to device during testing and allowing for a more fluid playtest.

To accomplish this we are going to create a new class that will contain our OSC functionality. Create a .cpp and .h file for this class now and name it LiveTesting. Open LiveTesting.h

And let’s add the line to import the OSC at the top of your file after your preprocessor directives and also a line for using iostream for testing purposes. Add the following code:

#include <iostream>

#include "ofxOsc.h"

Next let’s set up all of our variables we are going to use to receive OSC data and map it to in game values. Add the following code into your class.

class LiveTesting

{

public:

LiveTesting();

//a default c++ constructor

void setup(); //for setup

void update(); //for updating

ofxOscSender sender;

//you can set up a sender! We are going to use this network connection to give us some visual feedback of our current game values.

ofxOscReceiver receiver;

//this is the magic! This is the port on which your game gets incoming data.

ofxOscMessage m;

//this is the osc message your application gets from your device.

//these are the values we will be tweeking during testing

float max\_enemy\_amplitude;

int interval\_time;

float max\_enemy\_shoot\_interval;

bool triggerBonus;

};

Now let’s jump over to the LiveTesting.cpp file. In this file we are going to set up our network address and the ports we are sending and receiving data on as the first order of business. However to go any further we are going to need to do some housekeeping and install additional software. For OSC to work it will need a local wifi network to send the messages across. Note, this tactic may not work for a network outside of your own because often a sysadmin will stop this kind of traffic from being transmitted on a very public network. We suggest brining an Airport Express or similar with you so you can quickly and wirelessly establish a local network for playtesting.

For the purpose of this chapter and to allow us to create an experience that will work on both Android and iOS we are going to use a piece of software called TouchOSC from this URL: <http://hexler.net/software/touchosc>

The desktop software is free however the matching software for your device will be $4.99. As a mater of principle, we endorse building your own tools and you could easily build a second oF project to be your OSC sender and receiver on your mobile device. With that said, nothing beats TouchOSC for speed, ease of use and complete, platform independent flexibility. If you are someone who often moves between an iOS and Android device on both Windows and Mac, this tool will become indispensible to you. As a games designer it can open up possibilities like changing levels on the fly, updating game variables, adjusting for player feedback and adding new features into and taking them out of your game as it’s running. We highly endorse using it and support the continued advancement of the tool. You can also use it with music production tools like Ableton Live and it comes with great presets for things like DJing and mixing music live. Go to the app store of your device and purchase the mobile version now if you would like to continue down this route.

After we get all of the tools downloaded and installed. Let’s start setting everything up. You are going to need two bits of information. You are going to need to know the IP address of your computer and the ip address of your laptop. If you are on a mac, just open up your System Preferences. Go to the Network setting and click on your wifi connection in the left sidebar. On the right side it will display your IP address. You can also get this setting by opening up Terminal and entering in the command ifconfig. Terminal will list of every network that’s a possible connection for your machine from the past, even if it’s not currently active. For example, if you have ever connected your phone, it will be in the list with some flag and listed as inactive. Look for the connection that’s currently active. It will look something like this:

en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500

ether 60:33:4b:12:e5:3b

inet6 fe80::6233:4bff:fe12:e53b%en1 prefixlen 64 scopeid 0x5

inet 192.168.0.5 netmask 0xffffff00 broadcast 192.168.0.255

media: autoselect

status: active

The inet address is your current IP.

On windows, open the charms bar. In search type cmd and open the command prompt. Type in ipconfig. This information is much clearer than the data dump from terminal. The connected listed as your Wireless LAM adapter Wi-Fi will list your current IPV4 address. This is your IP address. Finally, obtain your mobile device’s IP address as well from your device settings.

Now we are ready to go back to our programming IDE. Open up LiveTesting.cpp

In our default constructor, we will now set up our game to send and receive values over the network. To do this we will need to know which Ip address and port on our device we will send to as well as set up a port on our local computer’s network to receive incoming data. Your computer will have only one IP address but it can send and receive data on thousands of ports. While we aren’t going too deep into ports there, you can think of the IP address like a boat pier. Lots of boats can be docked at a single pier. This is no different. Your ports are your docks and your IP address is your pier. You can think of the data like the people departing and arriving. You’ll need a separate port for each activity in this scenario. By and large if a port isn’t used by your operating system, you can send and receive data there. In this case we are going to use 8000 and 8001. The final thing we will establish is the Address Pattern**.** It will look like a file path and it will allow us to specify the address pattern match our messages to their right values. Add this code:

#include "LiveTesting.h"

LiveTesting::LiveTesting(){

sender.setup("192.168.0.11", 8000);

//this is the ip address of your ipad/android and the port it should be

//set to receive on

receiver.setup(8001);

//this is the port you're game will receive data on. For us this is the important one! Set your mobile device to send on this port.

m.setAddress("/game");

//this is OSC's URL like naming convention. You can use a root url address like structure and then everything under that address will be accessable by that message. It's very similar to a folder path on your hard drive. You can think of the game folder as your root directory and all the bits that are /game/someOtherName are inside of it.

}

//Left to do – explain touch OSC and the last function

//implement sprite sheet

1. . A few windows touch osc notes for windows users when I get this far.

   Make sure you install java from java.com before you get going (JavaSE runtime). The 32 bit version works with the windows bridge so just use it.

   Ipconfig is Ifconfig on a mac

   users are looking for the Default Gateway address to input into the touch osc hosts panel

   Formatting:https://github.com/openframeworks/ofBook/blob/master/styleGuide.md [↑](#footnote-ref-1)