CS50's
Introduction to Game Development

OpenCourseWare

Colton Ogden (https://www.linkedin.com/in/colton-ogden-0514029b/)
cogden@cs50.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 6: Angry Birds

Today's Topics

- Box2D
  - The physics engine we'll be using to build our game.
- Mouse Input
  - We'll discuss mouse input a bit more than we have in previous lectures, particularly since it is analogous to touch input, which is what the original Angry Birds mobile game relies on.

Download demo code

- github.com/games50/angrybirds (https://github.com/games50/angrybirds)

Sprites

- We use three sprite sheets in this program- one for our "birds" and "pigs" (round and square aliens, respectively), another for our world objects (ground, sky, flags, etc.), and yet another for our obstacles (walls, platforms, etc.).
  - For our obstacles sprite sheet, you'll notice that the sprites are not evenly laid out.
  - To get around this from a programming point of view, we had to hardcode each Quad's `x` and `y` coordinates to generate our quads properly.
  - This is a common strategy when dealing with sprite sheets whose sprites are of shapes that don't line up cleanly.
  - It's a bit of extra work to get started, but fortunately, you only have to do it once.

Useful Links

- Below are some useful links for learning more about Box2D physics:
  - love2d.org/wiki/love.physics (https://love2d.org/wiki/love.physics)
  - love2d.org/wiki/Tutorial:Physics (https://love2d.org/wiki/Tutorial:Physics)
  - iforce2d.net/b2dtut/introduction (http://www.iforce2d.net/b2dtut/introduction)

The World

- When creating a Box2D physics-based game, the first thing we will need is a system to perform the physics simulations (i.e., the world).
- The world performs all physics calculations on all "Bodies" to which it holds a reference.
- It possesses a gravity value that affects every Body in the scene in addition to each Body's own characteristics.

Important Functions

- `love.physics.newWorld(gravX, gravY, [sleep])`:
  - Creates a new World object to simulate physics, as provided by Box2D, with `gravX` and `gravY` for global gravity and an optional sleep parameter to allow non-moving Bodies in our world to sleep (to not have their physics calculated when they're completely still, for performance gains).

Bodies

- Bodies are essentially abstract containers that manage position and velocity.
- They can be manipulated via forces (or just raw positional assignment) to bring about physical behavior when updated by the World.

Important Functions

- `love.physics.newBody(world, x, y, type)`:
  - Creates a new Body in our world at x and y, with type being what kind of physical body it is (`static`, `dynamic`, or `kinematic`, which we'll explore).

Fixtures

- Bodies are shapeless by default; this is where Fixtures come in. Fixtures are the individual components of Bodies that possess physical characteristics to influence Bodies' movements.
- Their purpose is to attach shapes to Bodies, influencing collision.
- Fixtures have densities, frictional characteristics, restitution (bounciness), and more.
- Below are some examples of useful functions for creating shapes:
  - `love.physics.newCircleShape(radius)`
  - `love.physics.newRectangleShape(width, height)`
  - `love.physics.newEdgeShape(x, y, width, height)`
  - `love.physics.newChainShape(loop, x1, y1, x2…)`
  - `love.physics.newPolygonShape(x1, y1, x2, y2…)`

Important Functions

- `love.physics.newFixture(body, shape)`:
  - Creates a new Fixture for a given body, attaching the shape to it relative to the center, influencing it for the World's collision detection.

Body Types

- As we mentioned before, there are 3 main Body types, and we'll explore them in more detail below.

- `static`: cannot be moved as by applying force, but can influence other dynamic bodies (like the ground).
- `dynamic`: A "normal" body that can move around and interact with other bodies, closest to real life (like a ball).
- `kinematic`: Can move but not influenced by the interaction of other bodies; an abstract type of body that's suitable for certain environmental pieces (like indefinitely rotating platforms that defy gravity).

Important Code

- To see an example of a `static` Body, open up `main.lua` in the `static/` subdirectory. This is a program that creates a static body (a simple square) and renders it on the screen.
  - `love.load()`: sets up the graphic parameters, and creates a world, body, shape, and fixture.

```
function love.load()
    math.randomseed(os.time())
    love.graphics.setDefaultFilter('nearest', 'nearest')
    love.window.setTitle('static')

    push:setupScreen(VIRTUAL_WIDTH, VIRTUAL_HEIGHT, WINDOW_WIDTH, WINDO
        fullscreen = false,
        vsync = true,
        resizable = true
    })

    world = love.physics.newWorld(0, 300)
    boxBody = love.physics.newBody(world, VIRTUAL_WIDTH / 2, VIRTUAL_HE
    boxShape = love.physics.newRectangleShape(10, 10)
    boxFixture = love.physics.newFixture(boxBody, boxShape)
end
```

  - `love.update()`: defers to the world's `update` method, which calculates collisions.

```
function love.update(dt)
    world:update(dt)
end
```

  - `love.draw()`: renders the world onto the screen.

```
function love.draw()
    push:start()
    love.graphics.polygon('fill', boxBody:getWorldPoints(boxShape:getPo
```

```
    push:finish()
end
```

- To see an example of a `dynamic` Body, open up `main.lua` in the `dynamic/` subdirectory and you'll notice that the code looks almost exactly like that of the `static` example.
  - However, we specify on line 48 that the body we're creating is of type `dynamic` rather than `static`. This is reflected in the execution of the program, which results in our body falling off the screen due to gravity.
  - The above example is not particularly interesting, so take a look at `main.lua` in the `ground/` subdirectory.
  - As you might've guessed, this program adds ground to the world from the previous examples in order to better demonstrate the interactions between dynamic and static objects.
  - `love.load()` defines a few additional variables to add ground to our world:

```
world = love.physics.newWorld(0, 300)

boxBody = love.physics.newBody(world, VIRTUAL_WIDTH / 2, VIRTUAL_HEIGHT
boxShape = love.physics.newRectangleShape(10, 10)
boxFixture = love.physics.newFixture(boxBody, boxShape)
boxFixture:setRestitution(0.5)

groundBody = love.physics.newBody(world, 0, VIRTUAL_HEIGHT - 30, 'stati
edgeShape = love.physics.newEdgeShape(0, 0, VIRTUAL_WIDTH, 0)
groundFixture = love.physics.newFixture(groundBody, edgeShape)
```

  - `love.update()` still defers to `world:update()`.
  - `love.draw()` contains a bit more logic for rendering the ground to the screen:

```
function love.draw()
    push:start()

    love.graphics.setColor(0, 1, 0, 1)
    love.graphics.polygon('fill', boxBody:getWorldPoints(boxShape:getPo

    love.graphics.setColor(1, 0, 0, 1)
    love.graphics.setLineWidth(2)
    love.graphics.line(groundBody:getWorldPoints(edgeShape:getPoints())

    push:finish()
end
```

- To see an example of a `kinematic` Body, open up `main.lua` in the `kinematic`

subdirectory.

- In this program, we've added three spinning bodies to our world, such that our dynamic body falls onto them and is knocked sideways to the ground.
- The code for this behavior is mostly in `love.load()`, in which we've added a few lines to create and store three new kinematic bodies (with the same shape) in a table, as well as three new kinematic fixtures in a table, giving them an angular velocity.

```
...

DEGREES_TO_RADIANS = 0.0174532925199432957

...

function love.load()
    math.randomseed(os.time())
    love.graphics.setDefaultFilter('nearest', 'nearest')
    love.window.setTitle('kinematic')

    push:setupScreen(VIRTUAL_WIDTH, VIRTUAL_HEIGHT, WINDOW_WIDTH, WINDO
        fullscreen = false,
        vsync = true,
        resizable = true
    })

    world = love.physics.newWorld(0, 300)
    boxBody = love.physics.newBody(world, VIRTUAL_WIDTH / 2, VIRTUAL_HE
    boxShape = love.physics.newRectangleShape(10, 10)
    boxFixture = love.physics.newFixture(boxBody, boxShape)

    groundBody = love.physics.newBody(world, 0, VIRTUAL_HEIGHT - 30, 's
    edgeShape = love.physics.newEdgeShape(0, 0, VIRTUAL_WIDTH, 0)
    groundFixture = love.physics.newFixture(groundBody, edgeShape)

    kinematicBodies = {}
    kinematicFixtures = {}
    kinematicShape = love.physics.newRectangleShape(20, 20)

    for i = 1, 3 do
        table.insert(kinematicBodies, love.physics.newBody(world,
            VIRTUAL_WIDTH / 2 - (30 * (2 - i)), VIRTUAL_HEIGHT / 2 + 45
        table.insert(kinematicFixtures, love.physics.newFixture(kinemat
        kinematicBodies[i]:setAngularVelocity(360 * DEGREES_TO_RADIANS)
    end
```

```
    end
```

- We do, of course, add in an additional tidbit to `love.draw()` in order to render these new bodies on the screen.

```
love.graphics.setColor(0, 0, 1, 1)
for i = 1, 3 do
    love.graphics.polygon('fill', kinematicBodies[i]:getWorldPoints(kin
end
```

- For a look at a more interesting all around example to conclude our discussion on Body types, open up `main.lua` in the `ballpit` subdirectory.
- This is a program in which a large body of high density (a white square) collides with several smaller bodies of lower density (colorful balls).
- When the user presses the "space" bar, the collision is repeated.

Mouse Input

Important Functions

- `love.mousepressed(x, y, key)`
  - Callback that executes whenever the user clicks a mouse button; has access to the `x` and `y` of the mouse press, as well as the particular "key" (or mouse button).
- `love.mousereleased(x, y, key)`
  - The opposite of love.mousepressed, which is fired whenever we release a mouse key in our scene; also takes in the coordinates of the release and the key that was released.

angry50

- Let's take a look at the distro now.

Important Code

- Open up `main.lua` in `angry50/`
  - The main thing to note here is how we're monitoring for mouse input, similarly to how we've monitored for keyboard input in previous lectures.
- Now open up `StartState.lua` in `src/states/`. In our angry birds program, the start screen has animation going on in the background similar to our `ballpit` example. In this

case, there are 100 square aliens colliding with each other within an invisible container.

- `init()` initializes all the necessary variables for creating the world, the container walls, and the aliens.

```
function StartState:init()
    self.background = Background()
    self.world = love.physics.newWorld(0, 300)

    self.groundBody = love.physics.newBody(self.world, 0, VIRTUAL_HEIGH
    self.groundShape = love.physics.newEdgeShape(0, 0, VIRTUAL_WIDTH, 0
    self.groundFixture = love.physics.newFixture(self.groundBody, self.

    self.leftWallBody = love.physics.newBody(self.world, 0, 0, 'static'
    self.rightWallBody = love.physics.newBody(self.world, VIRTUAL_WIDTH
    self.wallShape = love.physics.newEdgeShape(0, 0, 0, VIRTUAL_HEIGHT)
    self.leftWallFixture = love.physics.newFixture(self.leftWallBody, s
    self.rightWallFixture = love.physics.newFixture(self.rightWallBody,

    self.aliens = {}
    for i = 1, 100 do
        table.insert(self.aliens, Alien(self.world))
    end
end
```

- `update(dt)` listens for a mouse click to transition to the PlayState (or for the "esc" key to be pressed to exit the program).

```
function StartState:update(dt)
    self.world:update(dt)

    if love.mouse.wasPressed(1) then
        gStateMachine:change('play')
    end

    if love.keyboard.wasPressed('escape') then
        love.event.quit()
    end
end
```

- `render()` displays everything on the screen.

```
function StartState:render()
    self.background:render()

    for k, alien in pairs(self.aliens) do
```

```
            alien:render()
        end

        love.graphics.setColor(64/255, 64/255, 64/255, 200/255)
        love.graphics.rectangle('fill', VIRTUAL_WIDTH / 2 - 164, VIRTUAL_HE
            328, 108, 3)

        love.graphics.setColor(200/255, 200/255, 200/255, 1)
        love.graphics.setFont(gFonts['huge'])
        love.graphics.printf('Angry 50', 0, VIRTUAL_HEIGHT / 2 - 40, VIRTUA

        love.graphics.setColor(200/255, 200/255, 200/255, 1)
        love.graphics.setFont(gFonts['medium'])
        love.graphics.printf('Click to start!', 0, VIRTUAL_HEIGHT / 2 + 40,
    end
```

- Move over to `Alien.lua` in `src/`:
  - `init()` initializes all necessary variables for creating an alien, taking in its world, type, coordinates, and additional data as parameters.

```
function Alien:init(world, type, x, y, userData)
    self.world = world
    self.type = type or 'square'

    self.body = love.physics.newBody(self.world,
        x or math.random(VIRTUAL_WIDTH), y or math.random(VIRTUAL_HEIGH
        'dynamic')

    if self.type == 'square' then
        self.shape = love.physics.newRectangleShape(35, 35)
        self.sprite = math.random(5)
    else
        self.shape = love.physics.newCircleShape(17.5)
        self.sprite = 9
    end

    self.fixture = love.physics.newFixture(self.body, self.shape)
    self.fixture:setUserData(userData)

    self.launched = false
end
```

  - `render()` displays the alien on the screen.

```
function Alien:render()
```

```
    love.graphics.draw(gTextures['aliens'], gFrames['aliens'][self.spri
        math.floor(self.body:getX()), math.floor(self.body:getY()), sel
        1, 1, 17.5, 17.5)
end
```

- Now that we have our aliens, let's examine `Obstacle.lua`:
  - `init()` sets up the obstacle's properties, keeping in mind whether it's vertical or horizontal.

```
function Obstacle:init(world, shape, x, y)
    self.shape = shape or 'horizontal'

    if self.shape == 'horizontal' then
        self.frame = 2
    elseif self.shape == 'vertical' then
        self.frame = 4
    end

    self.startX = x
    self.startY = y
    self.world = world
    self.body = love.physics.newBody(self.world,
        self.startX or math.random(VIRTUAL_WIDTH), self.startY or math.

    if self.shape == 'horizontal' then
        self.width = 110
        self.height = 35
    elseif self.shape == 'vertical' then
        self.width = 35
        self.height = 110
    end

    self.shape = love.physics.newRectangleShape(self.width, self.height
    self.fixture = love.physics.newFixture(self.body, self.shape)
    self.fixture:setUserData('Obstacle')
end
```

  - `render()` displays the obstacle on the screen.

```
function Obstacle:render()
    love.graphics.draw(gTextures['wood'], gFrames['wood'][self.frame],
        self.body:getX(), self.body:getY(), self.body:getAngle(), 1, 1,
        self.width / 2, self.height / 2)
end
```

Collision Callbacks

- The World fires four different functions for each collision between any two fixtures: `beginContact`, `endContact`, `preSolve`, and `postSolve`.
- Defining these callbacks allows you to program what happens at any given stage of any collision, beyond just objects bouncing off each other.
- Below is a link if curious to learn more:
  - love2d.org/wiki/Tutorial:PhysicsCollisionCallbacks (https://love2d.org/wiki/Tutorial:PhysicsCollisionCallbacks)

Important Functions

- `World:setCallbacks(f1, f2, f3, f4)`
  - `f1 (beginContact)`: fired when a collision begins.
  - `f2 (endContact)`: fired when a collision ends.
  - `f3 (preSolve)`: fired before a collision resolves.
  - `f4 (postSolve)`: fired after a collision resolves, with resolve data.

Level

Important Code

- Open up `Level.lua` in `src/`. This is the class we'll be using to instantiate the only existing level in our program.
- `init()`:
  - Creates a world and a table to keep track of destroyed bodies, then defines the four collision callback functions, leaving `endContact`, `preSolve`, and `postSolve` empty for now.

```
function Level:init()
    self.world = love.physics.newWorld(0, 300)
    self.destroyedBodies = {}

    function beginContact(a, b, coll)

        ...

    end
```

```
    function endContact(a, b, coll)

    end

    function preSolve(a, b, coll)

    end

    function postSolve(a, b, coll, normalImpulse, tangentImpulse)

    end

    ...

end
```

- Additionally sets the "bird" alien's `launchMarker`, creates tables for all aliens and obstacles in the level, and spawns the aliens and obstacles (as well as the ground).

```
function Level:init()

    ...

    self.world:setCallbacks(beginContact, endContact, preSolve, postSol

    self.launchMarker = AlienLaunchMarker(self.world)
    self.aliens = {}
    self.obstacles = {}

    self.edgeShape = love.physics.newEdgeShape(0, 0, VIRTUAL_WIDTH * 3,

    table.insert(self.aliens, Alien(self.world, 'square', VIRTUAL_WIDTH

    table.insert(self.obstacles, Obstacle(self.world, 'vertical',
        VIRTUAL_WIDTH - 120, VIRTUAL_HEIGHT - 35 - 110 / 2))
    table.insert(self.obstacles, Obstacle(self.world, 'vertical',
        VIRTUAL_WIDTH - 35, VIRTUAL_HEIGHT - 35 - 110 / 2))
    table.insert(self.obstacles, Obstacle(self.world, 'horizontal',
        VIRTUAL_WIDTH - 80, VIRTUAL_HEIGHT - 35 - 110 - 35 / 2))

    self.groundBody = love.physics.newBody(self.world, -VIRTUAL_WIDTH,
    self.groundFixture = love.physics.newFixture(self.groundBody, self.
    self.groundFixture:setFriction(0.5)
    self.groundFixture:setUserData('Ground')
```

```
    self.background = Background()
end
```

- `beginContact(a, b, coll)`:
  - The only collision callback we need to define in this example, since we already know what behavior we want to induce as soon as any two fixtures collide (recall that bodies do not collide, since they are abstract- this is done by fixtures).
  - In this case, we check which two fixtures have collided (player and obstacle, player and alien, player and ground) and define the appropriate behavior for each collision.

```
function beginContact(a, b, coll)
    local types = {}
    types[a:getUserData()] = true
    types[b:getUserData()] = true

    if types['Obstacle'] and types['Player'] then

        ...

    end

    if types['Obstacle'] and types['Alien'] then

        ...

    end

    if types['Player'] and types['Alien'] then

        ...

    end

    if types['Player'] and types['Ground'] then

        ...

    end
end
```

  - You'll notice that we *never* delete any bodies or fixtures within this callback

function- and with good reason!

- Deleting a body or fixture during a collision is a surefire way to introduce buggy behavior to your program.

- What we do instead is we flag any bodies or fixtures that need to be deleted and we delete them afterwards, when the collision has fully finished.
  - Keep in mind that deleting a body will delete all fixtures associated with it, whereas deleting a fixture will only delete that individual fixture.
- For example, this is how we destroy an obstacle when the player collides with it at a fast enough velocity:

```
if types['Obstacle'] and types['Player'] then

    local playerFixture = a:getUserData() == 'Player' and a or b
    local obstacleFixture = a:getUserData() == 'Obstacle' and a or

    local velX, velY = playerFixture:getBody():getLinearVelocity()
    local sumVel = math.abs(velX) + math.abs(velY)

    if sumVel > 20 then
        table.insert(self.destroyedBodies, obstacleFixture:getBody
    end
end
```

- `update(dt)`:
  - Updates the player's launch marker (i.e., the trajectory graphic) and the rest of the world, resolving collisions and processing the aforementioned callback functions.

```
function Level:update(dt)
    self.launchMarker:update(dt)
    self.world:update(dt)

    ...

end
```

  - It is only after the `world:update` function terminates that we delete the bodies/fixtures that have been flagged for deletion in the collision callbacks.
  - Since this function is called frame by frame, we reset the `destroyedBodies` table and remove anything from the screen that has just been destroyed in preparation for the next frame.

```
function Level:update(dt)
```

```
    ...

    for k, body in pairs(self.destroyedBodies) do
        if not body:isDestroyed() then
            body:destroy()
        end
    end

    self.destroyedBodies = {}

    for i = #self.obstacles, 1, -1 do
        if self.obstacles[i].body:isDestroyed() then
            table.remove(self.obstacles, i)

            local soundNum = math.random(5)
            gSounds['break' .. tostring(soundNum)]:stop()
            gSounds['break' .. tostring(soundNum)]:play()
        end
    end

    for i = #self.aliens, 1, -1 do
        if self.aliens[i].body:isDestroyed() then
            table.remove(self.aliens, i)
            gSounds['kill']:stop()
            gSounds['kill']:play()
        end
    end

    ...

end
```

- `render()` draws everything on the screen, as usual.

AlienLaunchMarker

- We saw in `Level.lua` that we generate the alien's launchmarker with a function call:

```
self.launchMarker = AlienLaunchMarker(self.world)
```

- Let's take a closer look at how this works.

Important Code

- Open up `AlienLaunchMarker.lua` in `src`. Here is where we'll find the code that allows us to launch our angry alien towards the obstacles.
- `init`:
  - Sets up the starting coordinates used to calculate the launcher's launch vector, keeps track of the shifted coordinates when clicking and dragging the launch alien, and creates flags for whether our arrow is showing where we're aiming, whether we launched the alien and should stop rendering the preview, for our alien which we will eventually spawn.

```
function AlienLaunchMarker:init(world)
    self.world = world

    self.baseX = 90
    self.baseY = VIRTUAL_HEIGHT - 100
    self.shiftedX = self.baseX
    self.shiftedY = self.baseY

    self.aiming = false
    self.launched = false

    self.alien = nil
end
```

- `update`:
  - Includes the logic for clicking and dragging the alien to show the launch vector and launching the alien when we release the mouse.
  - Once the mouse is released, the `self.alien` flag is instantiated with an `Alien` object, which is then provided with all the information regarding its coordinates, trajectory, and velocity.

```
function AlienLaunchMarker:update(dt)
    if not self.launched then
        local x, y = push:toGame(love.mouse.getPosition())
        if love.mouse.wasPressed(1) and not self.launched then
            self.aiming = true
        elseif love.mouse.wasReleased(1) and self.aiming then
            self.launched = true
            self.alien = Alien(self.world, 'round', self.shiftedX, self
            self.alien.body:setLinearVelocity((self.baseX - self.shifte
            self.alien.fixture:setRestitution(0.4)
            self.alien.body:setAngularDamping(1)
            self.aiming = false
        elseif self.aiming then
```

```
                self.shiftedX = math.min(self.baseX + 30, math.max(x, self.
                self.shiftedY = math.min(self.baseY + 30, math.max(y, self.
            end
        end
    end
```

- `render`:
  - Draws the alien and launch vector to the screen.
  - You can read more here about the math behind calculating the launch vector to draw:
    - iforce2d.net/b2dtut/projected-trajectory (http://www.iforce2d.net/b2dtut/projected-trajectory)
  - Needless to say, we stand on the shoulders of others and include within our `render` function the logic that others have already calculated.

```
function AlienLaunchMarker:render()
    if not self.launched then
        love.graphics.draw(gTextures['aliens'], gFrames['aliens'][9],
            self.shiftedX - 17.5, self.shiftedY - 17.5)

        if self.aiming then
            local impulseX = (self.baseX - self.shiftedX) * 10
            local impulseY = (self.baseY - self.shiftedY) * 10
            local trajX, trajY = self.shiftedX, self.shiftedY
            local gravX, gravY = self.world:getGravity()

            for i = 1, 90 do
                love.graphics.setColor(255/255, 80/255, 255/255, ((255
                trajX = self.shiftedX + i * 1/60 * impulseX
                trajY = self.shiftedY + i * 1/60 * impulseY + 0.5 * (i
                if i % 5 == 0 then
                    love.graphics.circle('fill', trajX, trajY, 3)
                end
            end
        end

        love.graphics.setColor(1, 1, 1, 1)
    else
        self.alien:render()
    end
end
```