Coding Report

DEMON PARASOL

Eric Gleiser – Technical Lead/Sound Lead BAGD

All content © 2014 DigiPen (USA) Corporation, all rights reserved.

GAM 205

NOTE: As mentioned, I will be doing a Coding Report instead of a Playtesting Report for this Milestone. Although this will be similar to a GAM 200 Student's Report, it will also focus on how the Zero Engine is being used in relation to the Game. It will also demonstrate the Zero Engines' Components interact with custom scripted components.

Technical Goals:

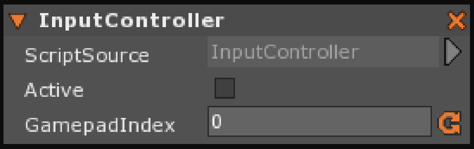
The Main Goal of any code written for this project is to make it easy for designers to understand how the games systems interact. This way adding content to the game is easy as all a designer has to do is tweak the properties in components to create data driven gameplay. It is also why we plan on using custom Enums instead of unreadable Integers whenever possible. An example usage of this can be found in the way footsteps work and create sound/water splashes.

Another goal is to have all components, other than those specifically for only the player or for debug purposes will avoid using the function "this.Space.FindObjectByName(name : String) Cog" to minimize dependencies, reduce null reference checks, and make the game run faster.

The final main goal is to make the code in the game mostly custom event driven. Although this may reduce speed slightly, it adds a lot of flexibility. For example this allows us to dispatch events containing data up and down hierarchies of trees. This is especially important for combat because it allows us to track what limb/part of an object has been hit. It also allows us to calculate the distance in object hierarchy trees for more advanced futures such as future implementations with the tongue mechanic.

Input:

class InputController : ZilchComponent (~53 Lines of Code)



InputController

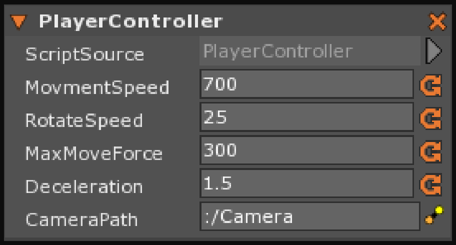
ScriptSource	InputController
Active	☐
GamepadIndex	0

The input is being handled by a script called **InputController** currently attached to the player. In the future this script may be attached to the LevelSettings Cog to make it more universally accessible. However the problem with attaching this component to the LevelSettings is the ability to toggle input acceptance per individual objects. This is why we will be attaching it to each individual object that requires input usage.

Luckily, the pointer to Gamepad object always points to the same object and is never duplicated to save on performance. This component automatically combines the Keyboard and Gamepad input and normalizes their vectors for seamless control switching. This component aims to unify all game input and also generates input vectors to be used by other controllers.

Player:

class PlayerController : ZilchComponent (~60 Lines of Code)

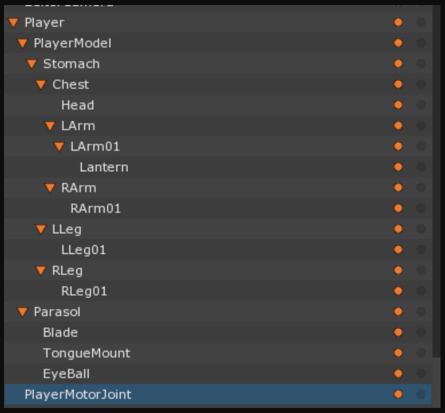


PlayerController

ScriptSource	PlayerController
MovmentSpeed	700
RotateSpeed	25
MaxMoveForce	300
Deceleration	1.5
CameraPath	:/Camera

The PlayerController class primarily handles the player's basic movements. It can adjust the movement vector based on the position of the camera. It also adjusts basic movement properties such as the speed, max move force, deceleration, and rotating speed.

Movement is calculated and sent using a DynamicMotor Controller that creates a Joint. This allows the PlayerController to create a constraint based physical Joint in which creates smooth motion for the player. This also allows the player to have a maximum force at which they moving making it impossible for the player to move certain objects depending on their mass. It also reduces the jitter often created by only setting the Velocity of an object. Because the DynamicMotor allows the player to be a constraint based RigidBody, this allows us to create Emergent Gameplay using various force effects in the Zero engine such as flow, buoyancy, and wind.



```
Player
  PlayerModel
    Stomach
      Chest
        Head
        LArm
          LArm01
            Lantern
        RArm
          RArm01
      LLeg
        LLeg01
      RLeg
        RLeg01
  Parasol
    Blade
    TongueMount
    EyeBall
PlayerMotorJoint
```

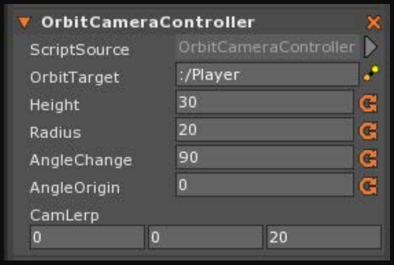
The PlayerController is also responsible for dispatching the attack event down to every object in the Player's hierarchy. This is what tells the animation to change and the currently held object (or potentially multiple object) to change to the attacking state. For example, the attack event would be dispatched down to the Parasol Object and the object's scripts will respond to the event.

class PlayerAnimator : ZilchComponent (~34 Lines of Code)

The PlayerAnimator handles all of the player's animations. It smoothly blends between the idle and walking animation and also will be linked to the FootstepDetector component. The speed of the animation is based on the velocity of the player. It interacts with the SimpleAnimator and AnimationGraph Components. Eventually more animations will be added as well as animation state control. The animations are partially data driven having an adjustable animation speed and hot-swappable animations that are then applied to a hierarchy of objects or the bones of a skinned mesh.

Camera:

class OrbitCameraController: ZilchComponent (~95 Lines of Code)



OrbitCameraController

ScriptSource	OrbitCameraController
OrbitTarget	:/Player
Height	30
Radius	20
AngleChange	90
AngleOrigin	0
CamLerp	0 0 20

This is the new camera for the game and is based on it orbiting around the player. It was created by Jason Clark AND myself. I spent a good deal of time walking Jason through this to help him become a better coder. This code is completely functional but we are planning to expand it to accept special CinemaEvents to make the camera more dynamic. Using Jason Clark's background in Cinema, we will create many useful event calls that make it easy to move the camera in interesting ways.

This camera is also reasonable for causing the game to flip. The flipping is disorienting to the player (as it breaks the traditional rule of 180 in cinema) and will ONLY be used when flipping between the Dream World and the Waking World. This flipping between worlds is never done multiple times within one minute. This camera is what renders the 3d objects in 2d space to meet the GAME200/205 criteria. It is also what the player's movement direction is based on so the controls do not become inverted when in a flipped game state. The camera is properly aligned to the game world by using an Orientation component. The position of the Camera is heavily Data Driven with Adjustable properties that can be accessed and changed in game.

class CameraController: ZilchComponent (~25 Lines of Code)

This was the original camera for the game. It was static and only interesting feature is that it follows its target smoothly by lerping the translation vector to an offset of the target. This Camera was ultimately scraped and was just created to get the game up and running quickly.

Also: The Camera has a MotionBlur post Process effect, Orientation, Microphone, and Fog components. The Camera's Projection is a 2D Orthographic projection.

Footsteps:

class FootstepSurface: ZilchComponent (~14 Lines of Code)

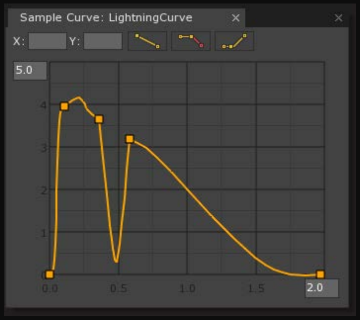
The FootstepSurface is applied to any object that can have a ray casted at it in the physics engine. It has an Enum property to set the type of material. This bit of data will be used for determining the type of sound, physics material, and footprint left by any object with the FootstepDetector Compnent attached.

class FootstepDetector: ZilchComponent (~88 Lines of Code)

The FootstepDetector component is applied to any object that has a reaction to the type of ground that is on top of. It works by casting a Ray downwards everytime the "Walk" function is called and reacting based on the cast results of that ray. The ray is casted into the physics space an returns the first object hit. It checks to see if the object hit contains a FootstepSurface before creating any footstep sound and will then play the appropriate sound based on the currently detected Enum from the FootstepSurface. It also has a timer system that is adjusted by animation speed to keep the sounds in sync with the animation. Also debug draws an arrow/line.

Effects:

class SampleCurveLight: ZilchComponent (~21 Lines of Code)



This component is to be applied to a Cog with a Light attached. It is designed to animate lights in a Data Driven way by sampling a SampleCurve resource over time. This time is adjustable with a local TimeScale. Its primary use is to animate flashes of lighting or flickering lights. This component allows for designers to easily animate lights with a cinematic mindset.

class Thunderstorm: ZilchComponent (~37 Lines of Code)

This component randomly generates strikes of lightning in the game. The chances of lighting are Data Driven with the DiceSize and LightingFreq properties. Every X amount of frames it will roll an n-sided die to determine if lightning should be created. This creates a clustered randomization technique. The Dice Rolls are generated using a random number generator object in Zilch It generates any archetype but is primarily used for lightening.

class WaterSplasher: ZilchComponent (~38 Lines of Code)

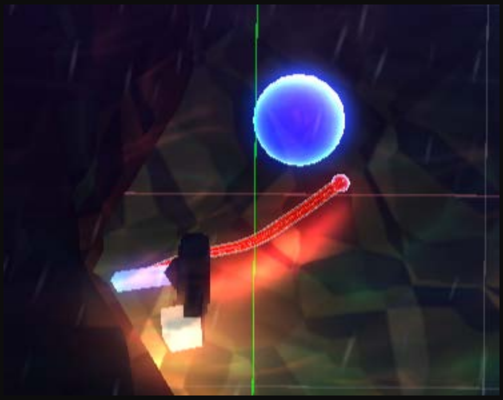
This component uses the FootstepSurface to determine if a body in water is floating or just entered the water. When it detects that a body has just entered the water it will generate a large splash sound and particle effect. While the object is floating, it will create particle based ripples in the water. This can be applied to any RigidBody from random floating crates to the player.

Combat:

class SwordController: ZilchComponent (~76 Lines of Code)

This component is attached to any weapon in combat. It makes the sword animate and is also responsible for timing combos. It creates a waltz-like microdynamic with its three step timing. It modifies the blade of the sword and only makes the blade collide during an attack by unghosting it. It also will dispatch events down and up the object hierarchy to the object it hit.

Shaders:



shader Fresnel Pixel fragment shader (~30 Lines of Code)

This Material Block Fragment Shader creates the Fresnel Rim-Backlighting effect. It is partially data driven and has the ability to make objects pulsate in time based on properties. (See the Ball in the Picture)

Total Code Count:

class InputController : ZilchComponent (~53 Lines of Code)

class PlayerAnimator : ZilchComponent (~34 Lines of Code)

class OrbitCameraController: ZilchComponent (~95 Lines of Code)

class CameraController: ZilchComponent (~25 Lines of Code)

class FootstepSurface: ZilchComponent (~14 Lines of Code)

class FootstepDetector: ZilchComponent (~88 Lines of Code)

class SampleCurveLight: ZilchComponent (~21 Lines of Code)

class Thunderstorm: ZilchComponent (~37 Lines of Code)

class WaterSplasher: ZilchComponent (~38 Lines of Code)

class SwordController: ZilchComponent (~76 Lines of Code)

shader Fresnel Pixel fragment shader (~30 Lines of Code)

~511

Lines of Code