Slash Framework Documentation

# Abstract

The design and rules of games constantly change during development, invalidating your carefully engineered software from day to day. Entity systems are a great approach for getting rid of the many drawbacks of inheritance-based game models like the “diamond of death”, moving on to a much more flexible aggregation-based model which has been popular since Gas Powered Games’ Dungeon Siege.

The Slash Framework provides both a low-level implementation of component-based entity systems and Unity3D integration for them.

# Game Breakdown

Implementing a game essentially boils down to three major parts:

* Rendering the game state
* Processing player input
* Updating the game logic

The Slash Framework leaves the first two parts to well-known game engines and focusses on providing data types for a clean, robust game logic model.

In fact, a Slash Framework *game* consists of the following parts that will be explained in the following sections:

* **Entity Manager.** Creates and removes game entities and entity components, such as trees or starships.
* **Event Manager.** Allows listeners to register for game-related events and notifies them whenever one of these events is fired.
* **System Manager.** Updates all game logic systems in each tick.

Creating and starting a new game is easy as this:

// Create new game.

var game = new Game();

// Start new game.

game.StartGame();

After a game has been started, you can cause a single tick to update the game logic, handle events and remove destroyed entities by calling the Update method:

// Tick game.

game.Update(dt);

# Entities

Each and every object in our game is called an *entity*. Entities can be visible or not, move, attack targets, explode, be selected by the player or follow a path. Thus, entities are common across games of all genres.

Defining specific game entities is done by adding *components*. Each entity component contains a part of the data that makes up that entity: An AttackComponent tells us how much damage is done by each attack of the entity, and the HealthComponent knows how much it can take before being destroyed, for instance.

Note that entity components don’t contain any game logic. Modifying the game state will be done by game systems discussed later.

## Defining Entity Components

In the Slash Framework, entity components are defined by implementing the IEntityComponent interface:

[InspectorComponent]

public class HealthComponent : IEntityComponent

{

/// <summary>

/// Attribute: How much damage the entity can take.

/// </summary>

public const string AttributeHealth = "HealthComponent.Health";

/// <summary>

/// Attribute default: How much damage the entity can take.

/// </summary>

public const int DefaultHealth = 25;

public HealthComponent()

{

this.Health = DefaultHealth;

}

/// <summary>

/// How much damage the entity can take.

/// </summary>

[InspectorInt(AttributeHealth, Default = DefaultHealth)]

public int Health { get; set; }

/// <summary>

/// Initializes this component with the data stored in the specified

/// attribute table.

/// </summary>

/// <param name="attributeTable">Component data.</param>

public void InitComponent(IAttributeTable attributeTable)

{

this.Health = attributeTable.GetIntOrDefault(AttributeHealth, DefaultHealth);

}

}

Here, the AttributeHealth constant is used as key for storing the health value in dictionaries and files. The Health property is the actual health value of the entity which is accessed and modified by game systems during the game. The InitComponent method is used for initializing the entity when it is created, using data read from an XML file, for example. The InspectorComponent and InspectorInt attributes are optional and used by Slash Framework editors we’re going to discuss later.

## Adding Entity Components

Other than in traditional game logic architectures, there is no game object or actor class in the Slash Framework. Game entities are nothing more than just a unique id. Actual game entities are created by mapping these ids to game components using an *entity manager*, like this:

// Create new game entity.

var entityId = game.EntityManager.CreateEntity();

// Create and attach new health component.

var healthComponent = new HealthComponent();

game.EntityManager.AddComponent(entityId, healthComponent);

Game logic will operate on these components by accessing and modifiying their properties through the entity manager:

// Get health component.

var healthComponent = game.EntityManager.GetComponent<HealthComponent>(entityId);

// Change health.

healthComponent.Health = 18;

# Systems and Events

As mentioned before, all game logic goes into game *systems*. There might be a physics system that is responsible for moving entities, or a health system that is changing entity health. All of these systems communicate by the means of game events, only.

Let’s take a look at an example: Say there’s a FightSystem that decides (in whichever way) that the entity with the id 337 should take 24 damage. It creates a TakeDamage event and sends this event to a dedicated event manager. At this point, the FightSystem is done, nothing more to do for it here.

Now, there’s a HealthSystem that has registered for this event, and asks the entity manager for the HealthComponent of the entity with the id 337. After having completed all computations, such as detracting armor or considering the damage type, it reduces the health value of the component by 19. After that, the system creates a DamageTaken event and hands that event over to the event manager.

There might be other systems as well, such as a SoundSystem that is interested in DamageTaken events for playing hit sounds, or an AnimationSystem that has registered for these events for playing hit animations.

There is a huge upside to this approach: There is no coupling at all between game systems. You can literally delete the AnimationSystem code file, and the game will be just fine with that.

## Defining Game Events

In the Slash Framework, each event has an *event type* and *event data*. You can easily define event types by creating your own enum:

public enum RPGGameEvent

{

/// <summary>

/// Entity health has been reduced.

/// </summary>

DamageTaken,

/// <summary>

/// Door has been closed.

/// </summary>

DoorClosed,

/// <summary>

/// Current player turn has ended.

/// </summary>

EndOfTurn

}

Some events don’t require more than a single value as data. The DoorClosed event from the above code snipped might be satisfied with just handing over the id of the door entity that has been closed. The EndOfTurn event might require no data at all. In case you need to pass more event data, just create your own data classes:

public class DamageTakenData

{

/// <summary>

/// Id of the entity whose health has been reduces.

/// </summary>

public int EntityId { get; set; }

/// <summary>

/// Amount of damage taken by the entity.

/// </summary>

public int Damage { get; set; }

}

## Creating Game Systems

Systems of the Slash Framework derive from the GameSystem base class:

[GameSystem]

public class HealthSystem : GameSystem

{

public override void Init(IAttributeTable configuration)

{

base.Init(configuration);

this.Game.EventManager.RegisterListener

(RPGGameEvent.EntityAttacked, this.OnEntityAttacked);

}

private void OnEntityAttacked(GameEvent e)

{

var entityAttackedData = (EntityAttackedData)e.EventData;

// Get damage done by attacker.

var attackComponent = this.Game.EntityManager.GetComponent<AttackComponent>

(entityAttackedData.AttackerId);

// Get armor provided by defender.

var armorComponent = this.Game.EntityManager.GetComponent<ArmorComponent>

(entityAttackedData.DefenderId);

// Compute final damage.

var damage = attackComponent.Damage - armorComponent.Armor;

// Reduce health.

var healthComponent = this.Game.EntityManager.GetComponent<HealthComponent>

(entityAttackedData.DefenderId);

healthComponent.Health -= damage;

// Notify listeners.

var damageTakenData = new DamageTakenData

{

EntityId = entityAttackedData.DefenderId,

Damage = damage

};

this.Game.EventManager.QueueEvent(RPGGameEvent.DamageTaken, damageTakenData);

}

}

The GameSystem attribute tells the Slash Framework to automatically add, initialize and update the system. Systems can use the Init method to register as listeners for specific game events. In the above code snipped, the HealthSystem is interested in EntityAttackedEvents. If such an event occurs, the system computes the actual damage caused by the attacker, reduces the target entity’s health, and notifies further listeners of the damage the entity has taken.

## Multiplayer, AI, Replays, anything else?

Note that this approach features further advantages: Moving the whole game logic to the server allows you to create multiplayer games with a minimum of additional effort. Player input events are sent to the server and processed there, while game logic events are routed to the client and used for updating the UI.

AI systems generate events just like player input would do. This way, the game logic doesn’t care whether input has been actually generated by players or AI – they’re processed in the exact same way.

Scripting support can be added to the game by causing game events through a debug console or cheat window.

Finally, storing all events along with timestamps allows you to easily create replay files for re-creating game experiences at a later time. Verbose log files can be created the same way.

# Entity Blueprints

Clearly, creating all game entities from code is not the way to go. We want our designers to be able to be creative, to invent awesome game mechanics and tweak each and every single component value. We’re going to need two additional concepts in order to achieve this.

Component values, such as the initial health of a knight in our role-playing game, are stored in *attribute tables*. In each of these tables, we’ll associate a key composed of the component name and the attribute name, with the respective component value. In most programming languages, these keys will be unique by language design.

Now, we can create *blueprints*, which are composed of a list of entity component types and an attribute table with values to initialize these components. Our fellow knight would have a component list containing a PositionComponent, a MovementComponent, a HealthComponent and an AttackComponent. His attribute table will contain his movement speed, initial health and attack damage.

Blueprints can be serialized to any arbitrary data format and made available for designers with custom editor tools. The designers can use these tools to create new blueprints, add components, and change all component values – without the need to re-compile the game!

<BlueprintManager>

<Entry>

<Id>Knight</Id>

<Blueprint>

<AttributeTable>

<Attribute keyType="System.String" valueType="System.Int32">

<Key>ArmorComponent.Armor</Key>

<Value>1</Value>

</Attribute>

<Attribute keyType="System.String" valueType="System.Int32">

<Key>AttackComponent.Damage</Key>

<Value>12</Value>

</Attribute>

<Attribute keyType="System.String" valueType="System.Int32">

<Key>HealthComponent.Health</Key>

<Value>40</Value>

</Attribute>

<Attribute keyType="System.String" valueType="System.Int32">

<Key>MovementComponent.Speed</Key>

<Value>35</Value>

</Attribute>

</AttributeTable>

<ComponentTypes>

<ComponentType>Logic.Components.ArmorComponent</ComponentType>

<ComponentType>Logic.Components.AttackComponent</ComponentType>

<ComponentType>Logic.Components.HealthComponent</ComponentType>

<ComponentType>Logic.Components.MovementComponent</ComponentType>

</ComponentTypes>

</Blueprint>

</Entry>

</BlueprintManager>

These blueprints can be loaded once when the game is started:

// Access blueprint file.

var blueprintFile = new FileInfo("Blueprints.xml");

using (var fileStream = blueprintFile.OpenRead())

{

// Deserialize blueprints.

var blueprintManagerSerializer = new XmlSerializer(typeof(BlueprintManager));

game.BlueprintManager =

(BlueprintManager)blueprintManagerSerializer.Deserialize(fileStream);

}

Given these blueprints, game systems can create the corresponding entities at run-time, for example at the start of each level:

// Get knight blueprint.

var knightBlueprint = game.BlueprintManager.GetBlueprint("Knight");

// Create knight entity.

var knightEntity = game.EntityManager.CreateEntity(knightBlueprint);

Finally, we are able to further configure our entities using *hierarchical attribute tables*. Say our level contains a whole army of knights. One of them is very unlucky and has been wounded in a previous battle. In the level editor, we create the wounded knight by adding an entity with the Knight blueprint. After that, we add an additional attribute table containing the new initial health value. This attribute table overrides the blueprint attribute table, replacing its values where applicable. We call this composition of a blueprint with an additional attribute table an *entity configuration*.

Then, each game has a blueprint file with common game data, while each level consists of a list of entity configurations, making up the specific game entities of that level.

// Assume this is read from an XML level file, for example.

var knightConfiguration = new EntityConfiguration();

// Get knight blueprint.

var knightBlueprint =

game.BlueprintManager.GetBlueprint(knightConfiguration.BlueprintId);

// Create wounded knight entity.

var knightEntity = game.EntityManager.CreateEntity

(knightBlueprint, knightConfiguration.Configuration);

Game designers love the new flexibility: A building that’s considered a tree? Don’t panic, just add the TreeComponent, and you’re done!