Overview Of Gameplay Systems And Their Interactions

Purpose

This document is intended to layout a high level overview of the pipeline for creating and editing game content. It intentionally avoids implementation details since the implementation is irrelevant, it will guide the descisions of the implementation.

Runtime Update Classes

Every entity that can be placed in the editor is an update class. Also, every entity that can be placed in the editor is a script node. Effectively script nodes and update classes are one and the same.

Update classes have data members that define their art dependencies. If a designer places a Hybrid, what he is placing is a node that will cause the hybrid update class to be built at run time. In the placed script node there are PVARs that define the mesh, rig, animations, effects, sounds, etc that are needed to build the run time art assets needed for a hybrid. If the designer changes the mesh and animations to a human mesh, then they have created a human that behaves like a hybrid.

At run time things might be implemented differently. Directional lights probably are best implemented outside of the update loop. However, they will need to be connected to the scripting system so that they can be turned on, off, brighter, dimmer, etc. To the editor user the optimization of the run time lights is inconsequential, therefore it should not be something that they need to work around.

DDL

The DDL is the center point to defining what goes into the game. Its purpose is to define the various permutations on how the data is presented. It is created and maintained by a programmer, and its inner workings are not exposed to the tool users.

Game Structure

The simplest form of a DDL entry is a structure that defines the data that is specified at build time. This is equivalent to the marking of PVARs in our current engine.

However, since every structure is a script note, every DDL will also include the list of inputs and outputs that a game object can handle. It may also include a list of links that the object needs. Finally it also specifies what runtime game object to associate with the data.

Generic Data Types

In some cases it is useful to specify a data structure that cannot be instantiated directly in the editor. As an example, assume that a game defined magic resistance as resistance to fire, water, wind and energy. This should be defined in a structure that is then used by every entity that can take damage. If the structure is changed at a later date, then it only needs to be changed once.

Components And Recursion

DDLs can specify components, where a component is more than a data structure; it is data, a mapping to some code, a list of script inputs and outputs, and a list of PVARs. Some of these elements might be left blank for a particular component. It is very similar to an update class; the main difference is that a component is never placed as a standalone entity in the editor.
An update class can define any number of components that it depends on. When associating components in the DDL, standard code bindings should be defined so that they may be automatically generated.

Editor Mapping

In addition to specifying the data, how to present and edit the data should also be laid out in the DDL.

Grouping Similar Data In The Editor

There are times when certain pieces of data may be logically grouped but defined separately in the DDL. For example, when defining the health for a Hybrid, it makes sense for the PVARs from a damage modifier component to be clustered with the health PVAR that was defined in the Hybrid DDL. The Hybrid DDL can contain information for use in the editor to map the PVARs into a presentation that differs from the order of declaration.
Likewise there are times when certain PVARs are defined, but should not be edited by the designers. As an example, if a Hybrid has a FSM asset that specifies the FSM to use for his behavior, it should be defined in the DDL so that the game automatically loads the FSM asset. However, the designers should not be changing the FSM. If a programmer develops a new FSM, they can update the DDL to point to the new asset.
Conversly, there are times when data is needed by the editor, but not by the run time. For example, a directional light may be placed in the editor 3D view so that the level builder may conveniently manipulate it. At run time the position is not needed. Similarly, a Hybrid needs to store its position in the 2D script view, This is data that should not be sent to the game run time. Especially since the hybrid may actually have a list of tens even hundreds of 2D script positions.
Finally, if the editor supports defining user profiles, then the various mappings can be tagged to be associated with certain profiles. This will allow Designer data to be hidden from Artists and vice versa.

Component Script Data

Since an entity may contain several components, which may in turn define script inputs, outputs, and links, the DDL needs to specify how those are exposed to the entity. For example if a Hybrid has three damage modifier components that all accept a take damage message. The placed Hybrid should have only one input, which maps to the three components. It should also be placed next to any other inputs that are health related.

Custom Data Editors

There are times when certain data types will require a custom editor. Some examples are:

Color Picker

Though color typically is represented as an RGB, there should be a custom editor that speeds up the data entry process. The Color data type should specify which editor to use in its DDL.

2D Spline

2D splines are amazingly useful parameters for game objects. They should appear in the PVAR list as a thumbnail showing their actual shape. When clicked a spline editor should open up allowing the user to adjust the spline length and shape.

Magic Resistance

Continuing the example given above, the design may well place restrictions on the values. For example, they may want the sum total of resistance to never exceed 200 * the character level. A custom editor would implement the logic to check that the data is always valid.

Custom Type Editors

In addition to certain PVARs needing custom editors, certain types will also need to specify custom editors. These may be specialized editors in the 3D world view, or in the 2D scripting view.

Math Node

A math node is one, which can deeply inspect any placed entity and perform math on its PVARs, the result of which may be applied to any entity. This requires it to be able to display the PVARs of its target source and destination.

Sequence Timer

A sequence timer is an entity, which by default has no outputs. The designer places one and defines a time period; then, they can define events within the time period, which then become outputs. When editing the sequence timer the designer needs to be able to scale the time line, add events, specify that certain events must happen exactly one frame apart (to handle harmonics that may happen when playing at 50Hz or 60Hz).

Path

A path is a 3D spline. Obviously a custom editor needs to handle placing and editing paths.

Emitter

If an emitter is placed in the world, the designer should be able to preview it. This will require a special render to find the effect specified in its PVAR data and display it according to its orientation.

DDL Upgrade Rules

Game creation is all about iterating on ideas. A common concern is how to handle the data associated with an update class that has changed. The changes may be added

PVARs; these are handled implicitly by the system. Similarly, deleting a PVAR can be handled with little effort. The only concern is for derived data that has been over ridden; it can either be deleted or stored in case the PVAR returns. Though the default case should probably be the safe “keep it”; sometimes deleting aPVAR might be because it is changing purpose and a reset on all set values is needed. To handle this, each entity should have a version, if a programmer increases its version number they should also provide an update rule. In the case provided it would be to delete the stored PVAR.
Since a placed script node is evaluated when a level is opened, the rules would be applied then. If assets are synced and the level is not opened before running the game, we would need to find a way to force upgrading all of the game data for the level. Note, this is true in all cases. If a user syncs to new data they we may need to force updating the game data on their machine.
Over the course of a project, new cases for upgrading data will be needed, as they arise script rules can be developed – which may well require creating functionality for the update rules.

Automatic Code Generation

The DDL defines a vast quantity of information that can be leveraged to automatically generate a large section of code. When a programmer modifies a DDL file, they should be able to process it into a CPP and H file. The file to generate and its path should also be part of the DDL. If that file already exists, the code generation should only change the sections that were appropriately blocked by comments indicating that the code was auto-generated.

Class Definition

As a start, the DDL defines a class structure. This can easily be auto generated. As part of the class, there are subclasses, whose H files are defined in their respective DDL files, so all C/C++ dependencies are known and can be leveraged. Components are also known, and their binding can be specified in the DDL, and built automatically.

Factory

The DDL also can result in the appropriate constructors to be defined. Serialization code, aka Factories, should be generated automatically.

Live Link

The code to handle live link data should be automatically generated. There may also need to be hooks to override the default behavior, as appropriate.

Message Handling

Since various components will expose their script inputs and outputs, the code to map them should also be generated automatically.

Programmer Customization

This automated code generation does not replace actual programming. All of the innate behaviors of an update class, and custom interactions with their components need to be implemented by a programmer. The generated code must specify through comment blocks which parts are safe for the programmer to edit.

Script Nodes In The Editor

Everything Is a Script Node

Since everything is a script node, the only function that the editor provides is the ability to place a chunk of data. It has no inherent types that it can manipulate directly other than a bucket of DDL data. Obviously, there are some types that benefit from custom support; this is achieved by specifying a custom editor or viewer as part of the DDL. Some examples include:

Lights

Placing a light in the editor would benefit from actually seeing the effect of the light on the level. The DDL specifying a light would also specify a plug-in for displaying the light that would achieve this.
By making a custom plug-in interface, the code for lighting can be applied to any placed object. Some project may desire to have a light in the editor to highlight checkpoints in a level; this would be trivially added without any additional programming.
Most importantly, when a light is first implemented by an programmer they can simply specify the PVARs needed in the DDL and go about writing the runtime code. When a tool programmer has time they can modify the DDL to specify the plug-in needed. All existing lights will now inherit this functionality without needing any change.

3D Splines

When placing a spline it would benefit from a custom render as well as a custom UI for editing it. Both of these should be specified as a plug-in in the DDL.

Trigger Volumes

Our current editor has custom code to handle rendering volumes. This should not be the case. If a wire frame box is desired, then a custom rendering plug-in can be written. This will allow a project to customize rendering of volume information. For example, a navigation animation clue may specify the wire frame rendering or it may specify a cube with a meaningful texture on it. Either way, this is simply DDL data that can be changed per type (possibly per instance even) and per project basis without any programing.

Default Values And Ranges

All data coming out of the editor should be valid. If invalid data is prevented at the editor level, we reduce the time it takes to find and fix errors. The DDL must specify a default value for every parameter, as well as a valid range.

Data Types

The DDL specifies data types for the various PVARs. The system is strongly typed. If a PVAR needs a damage effect, then only damage effects (as opposed to more generic sound or visual effects) can be selected.

This requires the asset system to support defining data types from all editors (e.g. Hybrid FSM vs. Door FSM, Hybrid Animation set vs. Grim Animation set).

Prefab Simple Type

The designers should have a list of all the update classes defined by the programmers. This is the list of base types that they can derive new types from. (Note, this is data derivation not C++ object derivation). The idea is that a designer can take a Hybrid and change the scale to define a tall hybrid. The Tall hybrid can be further derived to a strong hybrid.
The derived types only store the delta to its parent type. As a result if any value in the hierarchy is changed that was not overridden in the derived type, the new value will be applied to the derived type. As an example, if the base hybrid health is changed then the changes will also apply to the tall hybrid but would not apply to the strong hybrid since it changed the health value.
.

Prefab Scripts

Scripts prefabs are just an extension of the simple type prefab. When a script prefab is defined all that needs to be stored are the types and their over ridden PVARs. Note that links are just a PVAR with a different presentation. If a script prefab is created using the strong hybrid, then any changes to the hybrid, tall hybrid or strong hybrid will carry over to the strong hybrid in the script prefab, unless that change is in conflict with data specifically over written in the script.
It should also be noted that scrip prefabs are simply “syntactic sugar” to improve the workflow of level creation. They provide consistency of experience and automation for repeated setups. The data that makes it into the game has no prefab information since it is only an editor construct.
Also, script prefabs create a new type in the pallet for the designers to work with. It can be derived from to create further new types, with all the derivation rules applying to the new types.
If a designer wants to modify the links between script objects that are part of the prefab then they must first break the prefab back into its component pieces. This is because there would be no way to resolve a change in the prefab definition against the derived type.

Placed Instances

An instance placed in a level is simply an extension of this type derivation. This means that the level only stores the type and the delta to the type that was placed. In most cases this will result in only the type and position information being stored. When the editor loads the level it builds all the information for the placed entity by walking from the base to the top of the derivation tree overriding values as it does so. The same will need to be done when creating the game data. The game data is built by constructing the base types as defined by a programmer. The values are overridden as the derivation tree is unrolled; the game only ever sees the base types as defined by a programmer.

Update Dependencies

Using a similar editor to the visual script editor, the level editor will provide a view where any placed entity can be connected to another so as to specify update order dependencies. The editor will automatically prevent cyclical dependencies. The data will feed into the runtime dependency API. At runtime other dependencies can be added, and the editor specified dependencies could also be modified as needed. This is simply so that a designer can create a situation where explicit dependencies are known without needing programming support.

PVAR Types

Most PVAR types are an amalgamation of existing numerical data types (integers, floating point numbers, enumerations, etc). However, there are PVARs that will need custom editors. An asset PVAR will conjure an asset browser and be displayed as the asset name (possibly a thumbnail). Assets should be strongly typed so that only valid assets can be assigned to any given PVAR. A 2D spline would display as a thumbnail of the spline, and when clicked on would conjure an editor for the spline.
In keeping with the data driven model of the editor these types would be defined in a DDL, as would the plug-in editor associated with the type.

Asset Types Of Note

Several asset types are worth mentioning explicitly. As the tools mature others will likely become equally important.

Character Set

Characters follow a well-defined pattern of data needed. They need a model, a rig, a list of animations that the game can access by some form of identifier (e.g. run, jump, shoot). They also may specify events that the game needs (e.g. gun shoot), or simply events that feed systems to provide richness (e.g. footfall).
A Character Set tool should provide a way to group and visualize all these parameters. A programmer would define a character type and with it the parts that the game must have to function. If a new version of the character (Note, the term character is fairly lose, this would apply to an exploding barrel as well as a hybrid) is desired then a new data set can be created and assigned to the PVAR of the character.
While in the level editor, double clicking on the character set thumbnail would invoke the character set editor.

FSM

Finite state machines are used to solve a wide range of problems; most notably, AI. A FSM tool and accompanying runtime will be developed to solve the largest need: AI. Once the tool has reached a level of maturity it will be evaluated, and hopefully promoted to use for general-purpose FSM work – including FSMs created by designers with no additional programmer support.
The intent for the FSM system is to support common activities through data. This includes network synchronization and associating visual script messages and events to the various state transitions.

Streaming

Many assets that are steamed in can happen in an automated manner with the game not needing to be aware of when it happens. In certain cases the state of streaming will be an explicit concern to the developer.

If a script object is streamed in it can not be activated until all script objects it is connected to are streamed in. If this is not the case a script's behavior will be undefined, acting one way when the objects are partially loaded and another way when they are fully loaded. This scenario is one that a designer will not have tools to detect and will not have tools to fix. The structure of streaming will need to be laid out in the editor. One suggestion is to implement zones as a hierarchy. Every level will have a global zone and any subsequent zones are children of this zone. The hierarchy is of non-determined depth and varies from project to project as well as level to level.

When linking a script nodes across sibling or child zone hierarchies a relay node in the parent zone must be used. It will hold the message until the zone of the target is fully loaded. Obviously, a script node can be defined that forces a zone to load or unload. Since everything is a script node – zones are script nodes too.