User scenarios and milestone exit criteria for M0

Creating our M0 milestone the gameplay way. Ideally we want to follow these exact steps to create our milestone exit game, but since the tool chain won’t be complete we should at least manually simulate these steps to form input data files. If we abstract the file parsing well, then we can even use something like XML to make it easy to hand author the files until we switch over to full tool chain (and then update the loader code to move to whatever binary format we come up with).

1. Create game object templates:
   1. Block – static mesh visual, default material
   2. Dude – static mesh visual, default material
   3. Camera man – no visual, camera component
2. Author main gameplay screen
   1. Place an instance of the Block object, with a large scale in the X and Z directions. This acts as our ‘ground’
   2. Place an instance of the dude on top of the block (no physics yet in M0), have this instance sink user action notifications with a script action
   3. Create user action table for the screen
   4. Place an instance of the camera man in the scene. Have this instance sink user action notifications

Packaging of a game will come in a later milestone, so for now we’ll have to hard code (or pass in via command line) the initial, and only, screen to the GameApplication object.

Code which needs to be written to achieve the above scenarios:

1. GameApplication object
   1. Manages application lifetime, plane for eventually supporting full screen and windowed, resizable and fixed.
   2. Figure out suspend/resume scenario and how it interacts with the screen manager. Doesn’t need to be implemented in M0, but we should have this in mind and carefully design in a way which easily allows this in the future and for certain platforms.
   3. Manages the global game timer, which may be configurable in the future (not fixed 60fps and/or vsync-locked)
2. ScreenManager object
   1. Manages loading screens, which can be loaded async on a background thread. We need to fully think out synchronization issues with background loading. Do we prevent calling update/draw? How does that screen load resources? Do all of our various resources (GPU, Audio, etc…) support loading from multiple threads?
   2. We may want to consider scenarios where maintaining a small back stack might be useful. Probably don’t need to support it now, but let’s see if it’s important for any game scenarios.
   3. Handles transitioning from one screen to the next. Initially, we’ll probably do a flat out replace, but eventually we’ll want to support fancy transitions.
   4. Hosts the screens, meaning that it exposes the core host interface to the child screens, and manages their active/focus states. The host interface (which doesn’t need to be implemented on the ScreenManager itself) is passed into each screen during load and provides core system services. Basically, it’s how the screens can reach services outside of their sandbox.
3. Screen sandbox object
   1. The Screen object in the engine isn’t necessarily what you’d think of as the GameScreen that you author in the tool. This is the container for that screen, and it provides the sandbox. Resource managers, Script contexts, and other similar objects will be instanced per screen, so that scripts from one screen can’t cross communicate directly with another screen, unless they formally go through the host services.
   2. We’ll want to allow for some resources to be identified as long living, and not get torn down with each screen transition (load them into some other, higher level resource manager).
   3. The screen houses a single script context, which contains a main update entry point for the entire screen. Other objects may internally contain their own script actions and callbacks, which will be parented to the main script context (so that the developer’s ‘global’ script variables are truly global within the screen). This global context is where we insert some system level objects and methods for the scripts to consume.
   4. Each Screen contains its own RendererContext, which appears on the surface like a new renderer. However, while the scene rendered by this context is unique to the screen, the underlying resources and graphics device are shared by all renderer contexts. Think of RendererContext as a ‘view’ or ‘session’ into the renderer, so it has its own scene of objects to draw, and essentially owns the view while it’s active, but it doesn’t own/control aspects like render targets, screen resolution, etc… Those are services provided across the RendererCore, and are configured game-wide.
   5. The screen also provides some core services that were configured in the GameScreen, such translation of input to user action notifications according to the user action table, and providing the primary engine tick for the screen. Once per update, it calls into the GameScreen’s update script method.
4. Notification system (messaging)
   1. Many actions, such as input, will be triggered via notifications. We need the basics of the notification system in place.
5. GameObject and components
   1. We need to solve the game object and component design. It needs to support all of the features in our scenarios above, but also keep a mind to the scenarios in the other doc full of mock ups.
6. ScriptContext object
   1. The ScriptContext is essentially a container for a series of registered objects (exposed to the script via thunks) and script function entry points. An oversimplified interface for example purposes might look like this:
      1. CallScriptEntryPoint(int32 scriptEntryPointId);
      2. CallObjectScriptEntryPoint(ScriptableObject\* thisObj, int32 scriptEntryPointId);
      3. RegisterGlobalFunction(const char\* name, ScriptableMethod\* method);
      4. RegisterGlobalObject(const char\* name, ScriptableObject\* obj);
   2. In the sample above, scriptEntryPointIds are what are bound to handlers on objects, and when they need to be invoked, it gets done through the context as shown above.
   3. ScriptableObject is some property bag like object that is exposed to scripts. They can access any of the properties or methods on it through the property bag. For example:
      1. Int a = dude.CurrentWeapon.Ammo;
      2. dude.Attack();
   4. In these examples, dude and CurrentWeapon are scriptable objects. Dude’s property bag contains an entry for CurrentWeapon (by name), which points to another ScriptableObject (the weapon). That object’s property bag then contains a integer property called “Ammo”.
   5. Similarly, we have ScriptableMethods, which are just wrapper objects around a function pointer with a specific signature, and some minimal metadata which allows us to translate between script and the native function pointer.
7. RendererContext
   1. We need to determine the cleanest, minimal interface for a renderer session for each screen. The guarantee we give the screen is full screen access, so it should be able to Clear the screen, add and remove renderable objects in a session-specific ‘scene’, and invoke a render. We want all of this behavior completely contained in Screen, and not exposed to GameScreen, which should be oblivious to the renderer underneath.
8. RendererCore
   1. For M0, I’m going to put a very simple forward renderer while the deferred renderer matures. This is also a good forcing function to prevent us from building too much coupling between the renderer and other systems. We should be able to switch between the deferred renderer and the forward renderer fairly easily. The only gotcha that needs to be figured out is user-defined materials and shaders. I have a solution in mind which will allow us to switch the engine-provided materials with ease, but I haven’t figured out how to create a material and shader authoring system for game developers which hides the forward/deferred rendering piece yet.
9. Keyboard input
   1. We already have a basic keyboard object, but the interface may get cleaned up if we decide on codebase-wide convention/pattern refinement. The basic functionality and code will be the same as it is today, just the shape of the interface may change.

That should cover the high level pieces that need to be written for M0. I’ve glossed over a few important details, such as how/where we do actual file parsing for the data input to these systems. I’ve also left out some more details that I think will need to be figured out when we deep dive into the Screen object. This is really the workhorse for the entire game engine. This drives the game tick, runs physics simulations, coordinates visuals and audio, and feeds the renderer. There are a lot of nuances that we’ll probably evolve here as we go, but with a well-defined separation between game-logic and screen management, it should go smoothly.

The script system also deserves some more attention. Using a true scripting language is definitely not in scope for M0. We should design a native adapter class which behaves like a script, with some additional hooks to monitor things like marshaling overhead, number of script calls, namespace and name resolution, object resolution, and other useful metrics to help us not only bootstrap the script interface with some C++ ‘scripts’, but also keep a close eye on this part of the system to ensure it doesn’t become a bottleneck. I fully expect that the performance critical pieces of our code will be in the physics and renderer, so scripts should be negligible in comparison. There are also opportunities to parallelize some of these tasks.