|  |
| --- |
| Signal Studios |
| TS2 TDD |
|  |
|  |
| **Signal Studios** |
|  |
|  |
|  |

|  |
| --- |
| Describes the technical architecture of Toy Soldiers 2, as well as the supporting SigEngine code base and toolset. |

Table of Contents

[Scene Graph 1](#_Toc257995803)

[Structure 1](#_Toc257995804)

[Entity Types 2](#_Toc257995805)

[Resource System 4](#_Toc257995806)

[Resource Model 4](#_Toc257995807)

[File Sources 5](#_Toc257995808)

[Multi-Threading 6](#_Toc257995809)

[Resource Dependencies 6](#_Toc257995810)

[File Types 6](#_Toc257995811)

[Renderer 8](#_Toc257995812)

[Overview 8](#_Toc257995813)

[Renderable Entities 8](#_Toc257995814)

[Surface Shading 9](#_Toc257995815)

[Geometry 10](#_Toc257995816)

[Draw Call Submission 11](#_Toc257995817)

[Audio 13](#_Toc257995818)

[WWISE Integration 13](#_Toc257995819)

[Animation 14](#_Toc257995820)

[Overview 14](#_Toc257995821)

[The Skeleton Resource 14](#_Toc257995822)

[Animation Track 14](#_Toc257995823)

[Animated Skeleton 15](#_Toc257995824)

[Key-Frame Animations 16](#_Toc257995825)

[Procedural Animations 17](#_Toc257995826)

[Scripting 18](#_Toc257995827)

[Overview 18](#_Toc257995828)

[Binding Scripts to Entities 18](#_Toc257995829)

[Exposing Native Code to Script and Vice Versa 18](#_Toc257995830)

[Game Engine Extensions 19](#_Toc257995831)

[AI 20](#_Toc257995832)

[Overview 20](#_Toc257995833)

[Behaviors 20](#_Toc257995834)

[Pathfinding 20](#_Toc257995835)

[Destructible Environments 22](#_Toc257995836)

[Overview 22](#_Toc257995837)

[Creation Process 22](#_Toc257995838)

[Triggering Destruction 22](#_Toc257995839)

[Physics 22](#_Toc257995840)

[Networking 23](#_Toc257995841)

[Note About Scope 23](#_Toc257995842)

[Synchronization 23](#_Toc257995843)

[Lockstep 23](#_Toc257995844)

[Distributed Entity Ownership 24](#_Toc257995845)

[Client/Server 24](#_Toc257995846)

[And the winner is… 24](#_Toc257995847)

[Potential Networking Libraries 25](#_Toc257995848)

[Fallback Strategy 25](#_Toc257995849)

[Saved Games/Serialization 27](#_Toc257995850)

[Overview 27](#_Toc257995851)

[Memory Management 28](#_Toc257995852)

[Overview 28](#_Toc257995853)

[Main RAM Allocators 28](#_Toc257995854)

[Video RAM Allocator 29](#_Toc257995855)

[Multi-Threading 30](#_Toc257995856)

[Overview 30](#_Toc257995857)

[Rendering 30](#_Toc257995858)

[File Loading and Decompression 30](#_Toc257995859)

[Ray Casting and Spatial Queries 31](#_Toc257995860)

[Particle System Processing 31](#_Toc257995861)

[Animation Evaluation 31](#_Toc257995862)

[Audio Streaming 31](#_Toc257995863)

[Localization 32](#_Toc257995864)

[Specifying Languages/Locales 32](#_Toc257995865)

[Localization of Text 32](#_Toc257995866)

[Localization of Images 32](#_Toc257995867)

[Asset Pipeline 34](#_Toc257995868)

[Overview 34](#_Toc257995869)

[Automation 34](#_Toc257995870)

[Source Assets 34](#_Toc257995871)

[External Technologies 35](#_Toc257995872)

[Listing 35](#_Toc257995873)

[Coding Standards 36](#_Toc257995874)

[Overview 36](#_Toc257995875)

[Notation Conventions 36](#_Toc257995876)

[Risks 38](#_Toc257995877)

[Overview 38](#_Toc257995878)

[Networking 38](#_Toc257995879)

[Performance 38](#_Toc257995880)

Scene Graph

# Structure

The scene graph is the spine of the game engine. It provides a unified interface and single point of access for all entities in the game, static and dynamic, logical and spatial. In this sense, it acts much like a database, supporting a very limited set of query options. Its structure is really that of a tree: entities can have a single parent and multiple children. This layout facilitates nested transformational hierarchies of entities – e.g., if the parent moves, so do its children, etc. – as well as purely “logical” hierarchies, where adding or removing an entity means adding or removing all of its children.

In traditional scene graphs, often the parent/child relationship is used to form Bounding Volume Hierarchies (BVH). Unfortunately, the hierarchical and logical relationships of entities in the scene often have no bearing on an efficient structuring for spatial partitioning. For this reason, SigEngine employs a separate spatial partitioning structure to perform intersection and ray cast queries. This separation creates for better management of spatial entities, and means that our spatial queries perform independently of the hierarchical structure of the scene.

Another artifact of the classic scene graph is that of “updating” entities. Generally speaking, scene graphs have been used for rendering entities, which has been extended to updating entities in various contexts. Usually, this process involves traversing the entire graph and calling a virtual “Update” or “Render” method on the entities. This process can be wasteful, however, as there will often be many entities that do not require any per-frame processing. Additionally, when used for rendering, there is a strong likelihood of visiting many entities that are not visible, or, worse, entities that are not even “renderable”; what’s more, when submitting immediate-style draw calls during this visitation, the ordering of draws is likely to be very inefficient for modern-day GPUs, where state sorting is essential. For these reasons, entity updating structures, like the spatial partitioning structures, have been separated out so that entities are only added to relevant lists for processing. Additionally, the scene graph’s only interaction in rendering is in providing an interface for performing spatial queries, so that renderers can get access to only the entities that are both spatially and logically relevant to the current rendering pass.

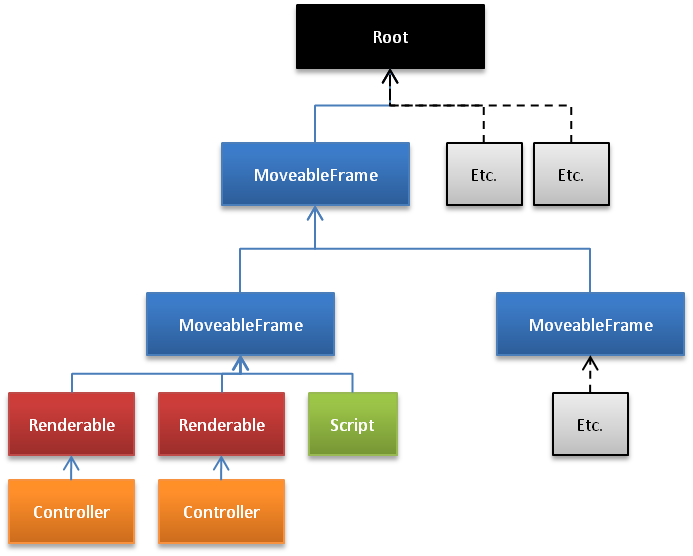
In an effort to minimize the number of global-style repositories in which entities are stored, both the separated spatial structures and updating structures mentioned above are owned and maintained by the scene graph, such that their existence is transparent to clients of the scene graph. Adding and removing entities to the scene graph for processing, updating, and rendering, is thus simplified for clients, as they do not also have to worry about adding and removing them from all the various other lists in which they should be tracked. Entities implement certain interfaces detailing their capabilities and needs, and so can be automatically handled by the scene graph on add and removal.

# Entity Types

The base Entity class provides the minimal services for existence as a node in the scene graph. These services include parent/child management, virtual methods for run list updates, as well as various lifetime notification methods (i.e., OnSpawn, OnDelete, etc.). It is worth noting that the base Entity class is a purely “logical” entity type (as opposed to spatial); it makes no assumptions about existence in the scene graph’s spatial set.

For spatial entities, there are two derived types providing extended functionality: Moveable Frame Entity, and Spatial Entity (which further extends Moveable Frame Entity). Moveable Frame Entities implement a single reference frame, as well as hierarchical transformations and notifications for children when moved. Spatial Entity is the base type for entities that actually have a bounding volume in the world, and hence is the base type used in the scene graph’s spatial partitioning structure.

The base renderable entity interface is Renderable Entity. Only Renderable Entities or derived types will be considered by the rendering system when performing frustum queries for each viewport. Renderable Entity exposes information required for insertion into Display Lists (see Renderer section, where the Renderable Entity and derived types are discussed in more detail).



Sample of potential sub-trees and entities within the scene graph

## Run Time Conversion Between Entity Types

Due to the nature of the various entity class hierarchies, game and engine code relies on being able to safely and efficiently cast between entity types. To this end, the Entity system implements a simplified equivalent to the C++ “dynamic\_cast” operator; given a pointer or reference to an entity, a client can attempt to query for a derived type by issuing a virtual method call using the run time type identifier of the target type. The query call will return NULL if the given entity does not actually implement the specified derived interface, and client code must always account for this result (i.e., checking for null before proceeding).

This custom RTTI solution is used so that classes can selectively include RTTI information, without requiring native C++ RTTI enabled for the entire source code base; additionally, classes have the ability to control which other types they can legally be cast to, without necessarily requiring that they inherit from the target cast type (i.e., the cast method can return the address of a member, or hide specific base classes).

## Entity Updating – Run Lists

The game update loop is separated into various stages in order to facilitate what can be referred to as “fork-and-join” multi-threading. Instead of the more traditional “Update” function, where a single entity will perform all of its work for the entire frame in a single function call, tasks are separated so that task batching can be optimized over multiple threads.

For instance, instead of one monolithic update function:

void Update( )

{

Animate( );

Move( );

FindTargets( );

ShootProjectile( );

}

each of the sub-tasks would be executed for ALL entities before progressing to the next sub-task. This allows for tasks that do not modify shared state to be executed in parallel (fork), minimizing the bottlenecks to stages where shared state is necessarily modified (join).

Resource System

# Resource Model

Resources in the game come in different flavors. At their most basic level, however, in terms of file I/O routines, asynchronous streaming, and decompression, we use a unified resource model, meaning that all file-based content is treated the same – basically just a raw stream of compressed or uncompressed bytes.

This unified model allows us to centralize the more common aspects of resource management and abstract it away from game code. Once the file is loaded into memory, however, we have allowances for treating files differently. Fundamentally, we have two distinct resource types: external file types and native load-in-place file types (discussed in more detail below).

Resources are acquired in the game by first querying a Resource Depot object. The game will utilize a single, shared Resource Depot object for all principal resources, though it is conceivable that separate Resource Depots could be used for the purpose of memory segregation, e.g. for facilitating level loads/unloads. Resources are treated as shared pointer objects, using smart-pointer reference counting semantics.

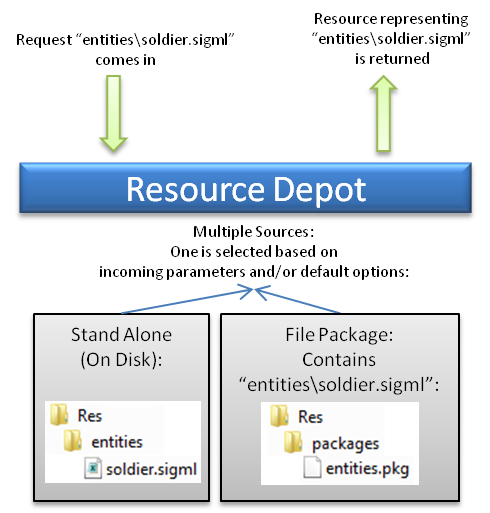
Acquiring a resource pointer does not actually cause a load of that resource; similarly, releasing a reference to a resource does not cause an unload of that resource’s file data. Loading and unloading is only initiated explicitly via a separate reference counted mechanism. “Load owners” are responsible for actually initiating these load/unload actions, in order to separate the roles of resource *use* and resource *management*. By maintaining a separate reference counting mechanism, multiple “load owners” can still safely share ownership and management of a resource. An example of the process is the following:

1. Object A requests a load for Resource R; Object A is now considered a “load owner” for Resource R.
2. Resource R is loaded.
3. Object B now requests a load for Resource R, and hence becomes a second “load owner” for Resource R.
4. Resource R, having already been loaded, does nothing.
5. Object A now requests to unload Resource R. Because Object B still maintains a load reference on Resource R, nothing happens, other than Object A is removed from Resource R’s list of load owners.
6. Object B now requests to unload Resource R. Because it is the last load owner, Resource R’s file data is now actually unloaded from memory. However, Resource R itself (a small, lightweight wrapper object) remains in existence until all smart pointers to Resource R go out of scope.

These ownership semantics imply that a client of Resource R may need to dynamically assess whether the file data itself is loaded or not. The benefits of this system however is that loading and unloading of resources is very easily controlled; rather than having to track down all references to a given resource, it is only necessary to track down the “load owners,” which are created by specific function call invocation, rather than object acquisition.

# File Sources

Resources are uniquely identified by a key derived from the underlying file’s path. However, the resource system actually enables the possibility of multiple sources for a single file. These potential sources currently include (but are not limited to) the actual physical file itself at the specified path, as well as a copy of the file embedded in any file packages that have been loaded.



Multiple disk-sources for resources: file packages vs. stand-alone file sourcing

The purpose of enabling multiple file sources is to ensure that for the released game, individual loose files can be bundled and packaged and then removed; by explicitly loading the file package containing those files, they will still be accessible in a completely transparent way to the acquirer of the resource.

During development, however, it is time-consuming and error-prone to force designers and artists to constantly re-bundle their loose assets, and so the resource system will first look for the standalone files and prefer these to any equivalent resource registered via a file package.

One additional benefit of this system is the added flexibility when it comes to downloadable content added after the game has already been installed. With small modifications, alternate sources can be added, allowing for multiple versions of content to exist on a user’s machine, without destroying the original set of files.

# Multi-Threading

Resource loading is fundamentally an asynchronous mechanism. All resource loading and decompression occurs in a secondary thread, using asynchronous, un-buffered file IO, reading “ideal”-size chunks of data from disk at a time (“ideal” being OS specific and subject to documented specs and empirical findings). While waiting for asynchronous loads to complete, this same resource loading thread will either block if the file is not compressed, or else decompress that block into the output buffer if the file is compressed.

Despite being fundamentally asynchronous, clients of resources can request to block until a resource is loaded. In most applications, however, blocking is undesirable and will be avoided. Resources provide a delegate-style callback mechanism to inform any interested clients when a resource load is complete; this callback will be executed in the main thread (the thread from which resource load was requested).

# Resource Dependencies

The resource system maintains a notion of “sub-resources,” or resources that are dependent on another resource, or in some way required and/or referenced by that resource. In general, initiating a load on a resource will also initiate loads on all sub-resources, and so on recursively until all resources required for the top-level resource have been loaded. Resource load callbacks will not activate until after all dependent resources have loaded. These dependencies are baked into resources and dependency information is created automatically by the tool chain. This process avoids programmers needing to be aware of the dependencies of a given resource, facilitating game code significantly.

# File Types

## External

“External file types” is the catch all for any file format defined externally to the game’s code base. Generally speaking, these files are relayed to a middleware package via our resource management system; this data would be considered opaque or blind data as far as the game and engine is considered, the resource manager simply acts as a pipe from file to main RAM.

It is worth noting however that though the engine supports this type of file, we currently employ no instances of it, as any 3rd party file types that we use are actually bundled into package files which are recognized directly by our engine.

## Native Load-in-Place

All file types defined in the engine and game use what is commonly referred to as “load-in-place” style serialization. Load-in-place simply means that data structures are written out to file in the tools pipeline into a block of memory such that the file can be reloaded into an identically-sized block of memory – all that is needed after loading the file is to apply “pointer-fixups”, such that any pointers written to file now actually point to the proper place in RAM after loading. Pointers are written to file as offsets from the beginning of the file, and converted back to pointers using the base address of the file’s memory block. Load-in-place style files can then be relocated in RAM using a similar technique, which can be useful for runtime defragmenting. While runtime defragmentation is not currently employed, it is a feature that could potentially be triggered explicitly between levels in order to move objects that for whatever reason are splitting large heap chunks.

One common problem with load-in-place serialization is that the serialization code, when multiplied over a diverse array of data structures, can be tedious and error prone to write and maintain. To account for this drawback, we employ a pre-processing tool which automatically parses tagged data structures and generates the serialization code for those types. The serialization code amounts to small reflection meta-data that becomes associated with the tagged classes and structs. Serializing an instance of a class is then reduced to calling a function on a block of data, specifying the reflection structure for that instance’s type; all type instances contained within the block, whether pointed to or directly embedded, can then be tracked automatically by the generated reflection code.

The benefits of this system (load-in-place combined with auto-serialization) are twofold: simplified code maintenance for both asset generation and file loading, as well as faster load times for assets. No parsing or other post-load processing is required for files: just load in the block of data, fixup pointers, and the data structures are ready to use.

The system is capable of handling types with virtual tables, cyclic pointer chains and non-duplication of data in the case of multiple pointers to a single “pointee”. Using this system, there is no need for boiler plate save/load code for our file types; we simply define data structures and tag them as “reflected”; these types can then be serialized to and from binary streams.

Renderer

# Overview

The Rendering System is comprised of many parts, with multiple systems running from tools to the game runtime. Its principal components are:

* Renderable Entities
* Surface Shading
  + Shaders
  + Materials
  + Lighting/Shadows
* Geometry
  + Static
  + Dynamic
* Draw Call Submission
  + Renderable Object Acquisition (Spatial Queries)
  + Draw Call Sorting
  + Draw Call Submission

# Renderable Entities

While traditional scene graph architectures might employ a virtual Render method, the game engine separates and encapsulates the rendering logic away from entities to a large degree. Nonetheless, there are common operations associated with making an object in the scene “renderable,” and as such we utilize a common base Renderable Entity class to facilitate these operations for game programmers.

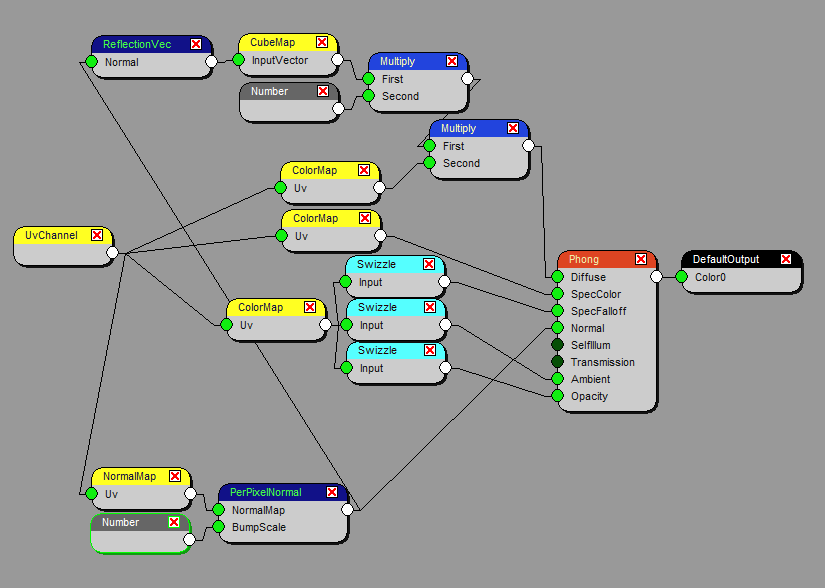
The Renderable Entity is a derived Spatial Entity, which means that it has volume and participates in the scene graph’s spatial set; specifically, the Renderable Entity maintains bounding information for its specific draw call.

Additionally, the Renderable Entity exposes the Render Instance and Render Batch data for insertion into relevant Display Lists (see below in the Draw Call Submission section for descriptions of these types). In this sense, the Renderable Entity’s primary purpose is to act as a vessel for the core renderable data structures – it provides spatial context and storage for them, and then hands them off to the lower level renderer for draw call submission.

# Surface Shading

## Shaders

Shaders are the templates for the artists’ materials. They provide the means by which our artists can define the unique look and feel of the game. As such, they are an extremely vital piece of the rendering pipeline. For this purpose, the shader architecture of the game engine is designed to provide maximal artist expressiveness and efficiency. By using a graphical node-based shader editor, artists can create and edit shaders without intermediate updates or builds from programmers. The shader editor is capable of exposing to artists all the raw functionality of the underlying HLSL code that it generates. However, to simplify the editing experience, small “nuggets” of common routines can be bundled and exposed as individual nodes to the artist. Artists can select lighting models, control mapping techniques, and create surface effects based on time and other dynamic variables directly in the editor.



An example screen shot of the shader node networks authored in SigShade

The shaders that the artists create in the editor are treated as first class resources, and so are easily tracked by the resource system as dependencies of materials.

## Materials

As mentioned, shaders act as templates for materials; the shader system is explicitly designed to promote re-use of shaders between many different materials by exposing parameters that the materials instantiate.

Examples of these parameters include fixed number values, color values, texture sources, and dynamic value bindings (e.g., time). These bindings are stored on disk with the material, separate from the shader. When binding a material to a mesh in the modeling package, artists select a shader as the basis for the material; at that point, a graphical interface for specifying parameters is automatically populated from the shader.

Materials are written to file along with the meshes they are bound to. At runtime, materials are responsible for binding their associated shader to the graphics device, as well as binding their parameters to the shader.

## Lighting/Shadows

All lighting in the game is computed in the pixel shader. Geometric properties required for the lighting equations (e.g., position, normal, tangent) are transformed into a “eye-relative world space,” which is to say a space with its origin at the camera’s eye, but with axes aligned with the world. The purpose of this space is to preserve precision for objects that are closest to the camera’s near plane, while avoiding some of the pitfalls associated with needing to transform vectors to view space. Additionally, the computation for transforming a position from world-space to this “lighting” space is simply a vector subtraction.

Lighting is applied using the more traditional “forward” rendering of geometry (as opposed to the newer “deferred”) – however, the lighting calculation is applied all in one pass, rather than being layered additively with multiple passes over geometry. To perform the lighting in one pass, all lights affecting a single object (draw call) must be gathered and passed to the pixel shader. This process of matching lights to objects is performed on the CPU during the “renderable object acquisition” phase (see below).

Lighting models are implemented as specific shader node types that the shader authors can select in SigShade, allowing for any number of different lighting models to be combined in the scene at a time. For example, rough metal materials or hair might choose an anisotropic lighting model, whereas a matte diffuse shader might choose to use the Oren-Nayar model.

# Geometry

Geometry that is rendered in the game generally comes in two flavors, static and dynamic. While these concepts can be a bit fuzzy, when we say static we actually mean geometry that is generated offline and stored on disk. Dynamic geometry on the other hand is created at runtime, in one of various ways.

## Static

Static geometry originates in either the artist’s modeling package, or else as terrain or other geometry created in the scene editor (SigEd). The attributes of the vertices stored within a given set of geometry are determined by the shader of the material applied to that geometry. For instance, if the shader performs normal mapping, then the vertices of the geometry will require tangents; similarly, if the shader requires a vertex color, then the geometry will include it. This determination occurs during asset conversion; for convenience, all possible vertex attributes are exported from the modeling package into the “intermediate” scene/mesh file format, and then only the relevant attributes are kept as the asset is converted to cooked, binary form for the game.

## Dynamic

Dynamic geometry is produced in-game, at run time. Examples include particles, decals, text glyphs, etc. To maximize efficiency of allocating this geometry, as well as for the purpose of providing upper bounds, we use fixed size buffer allocators from which we service smaller, individual geometry allocation requests.

Dynamic geometry is bound to materials and shaders in much the same way as static geometry – the process simply happens at run time. Producers of the geometry therefore have to ensure that they adhere to the proper vertex layout.

# Draw Call Submission

## Renderable Object Acquisition (Spatial Queries)

Before entities can be rendered, we must first determine which ones to render. To this end, each active viewport performs a frustum query of the scene graph, retrieving all visible and active Renderable Entities, as well as all lights whose volumes intersect the frustum (and hence all lights that can potentially affect objects in the frustum). Given these lists, lights are matched to entities, again bounding volumes so that lights can be trivially rejected from affecting entities.

## Draw Call Sorting

Once the visible entities have been determined and lights have been bound to them, the parts that are actually required for rendering are extracted and sorted.

The actual process of submitting draw calls to the graphics card is broken down using three essential data structures: the Render Instance, the Render Batch, and the Display List. Together, these types form the basis for efficient, GPU-friendly sorting of draw calls by state (shaders, textures, render states, etc.).

Objects are sorted differently depending on the Display List they are inserted into. For example, opaque objects are sorted using the Render Batch hash as its principal sorting criterion, so that all instances with identical Render Batch will be rendered together. Transparent objects, on the other hand, require back to front sorting using their depth from the camera’s point of view.

## Draw Call Submission

With all the Display Lists having been sorted, they can now be processed for actual draw call submission. This process is broken down into the application of batch state, and then the application of instance state. The first time a new batch is encountered, its state is applied (material, render states, shaders, geometry, etc.). From that point on, only the individual instance state need be set (world transforms, instance tint, etc.). Depending on usage and setup in the scene, this prevents a large amount of redundant state setting, as all instances sharing the same Render Batch will be adjacent in the Display List. Even in the case of transparent objects, where instances are sorted back to front, there is the potential that some of the instances that are adjacent in the list will share the same Render Batch; the Display List will automatically detect this case and only apply the first Render Batch in the group.

Audio

# WWISE Integration

We intend to use the battle-tested WWISE SDK for all audio play back. The comprehensive toolset, coupled with the object-oriented event system make the audio platform appealing for both sound designers and gameplay audio integrators.

Animation

# Overview

The animation system discussed in this section applies to skeletal-joints that can be rotated, translated, and scaled. Potential targets for this type of animation include skinned, deformable meshes, as well as rigidly bound attachment points. While the primary source of such animation is keyframe animations created offline in the modeling package, the system also supports procedural animations, as discussed below.

# The Skeleton Resource

All skeleton instances in game (“Animated Skeletons”) reference a skeleton resource file. Skeleton resources are created by animators in the modeling package and exported to a custom skeleton file format. It is at this point in the pipeline that the overall bone hierarchy is established, and bone names and master indices are also created. Any keyframe animation files created by the animator will be subject to validation and matching against their corresponding skeleton files.

Because skeletons exist as stand-alone resource files, they are easily shared by multiple meshes – additionally, they are free to be bound to multiple animation packs. This framework provides flexibility for animators to drive multiple meshes off identical skeletons and a common set or sub-set of animations.

Entities are made to animate by binding skeleton resources to them in SigEd; this binding is exported as part of the scene graph file. When bound in this way, a new Animated Skeleton instance will be created and attached to any sub-meshes as the scene graph file is loaded at runtime in the game. Control of the Animated Skeleton instances is then handled via script and game code.

# Animation Track

Animation tracks act as the basic unit of animation processing. The Animation Track class itself is merely a base class, implementing common functionality but leaving the primary implementation to derived classes. The derived animation track is expected to provide two basic services: stepping, and evaluating (i.e., it is expected to override these abstract methods).

The act of stepping an animation track entails performing any non-visual logic associated with the advancement of time. In most cases, the two most important actions associated with stepping an animation forward is motion extraction and event firing (see below for explanations of both). Evaluation, on the other hand, is the action of computing the bone transformations at the current moment in simulation time, resulting in a matrix palette which can be used for skinning bound meshes and rendering.

Animation tracks support various options, including time scaling (negative and positive values), range clamping/wrapping, pausing/unpausing, and blending in/out (looping vs. one-shot status). Additionally, animation tracks are not restricted to operating on the whole of the skeleton or any subset in particular; it may apply to a single bone, all the bones, or anything in between. It is up to the implementation of the derived Animation Track type to decide how it sources its bones. In fact, derived Animation Tracks are free to act as a container for multiple Animation Tracks, enabling blend-tree functionality.

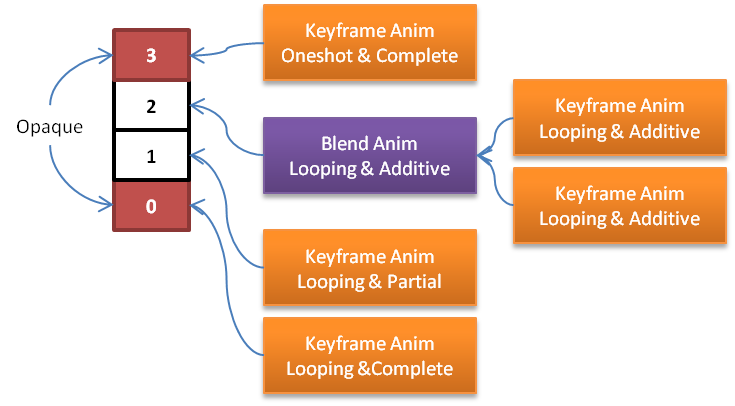
# Animated Skeleton

Animation Tracks are bound to entities at runtime using instances of the Animated Skeleton type. The Animated Skeleton’s fundamental purpose is to manage track-based layering and blending over time, in ways that resemble a stack. The Animated Skeleton provides an interface for “pushing” animation tracks to the top of its track list.

The way animation tracks are blended together follows a fairly simple rule set:

1. The higher the track is in the stack (the more recently it was pushed), the higher its blending precedence. E.g., if the last animation pushed were at 100% and being blended with standard blending semantics (“lerp” blending, as opposed to additive) then no track beneath it in the stack will be visible.
2. Tracks that are not blending in and are at 0% strength are removed from the stack.

Together, these fairly simple rules combine for an effective mechanism for layering animations on top of each other. Assuming a base animation track that’s looping, a client of an Animated Skeleton can push a one-shot animation and have it automatically revert to the previous looping track when finished; additionally, multiple partial tracks (animation tracks covering a sub-set of the entire skeleton) can be added without clearing the stack, meaning an animator could compose a complete animation using re-usable partial animations.



Sample view of a potential animated skeleton

The Animated Skeleton will accumulate tracks (ordered from least-recently pushed to most-recently pushed) until a looping “opaque” track is pushed, at which point the stack is cleared and the opaque track becomes the new bottom of the stack list. An “opaque” animation track is one which fulfills the following criteria: evaluates all bones (complete, as opposed to partial) and is blended in at full strength using lerp-style blending. When this criteria is met, and the animation is looping, no tracks beneath the opaque one will ever contribute in any way to the final animation result; hence, tracks beneath such a track are “invisible,” and become cleared.

# Key-Frame Animations

Key-frame animations are the brick and mortar of the animation track types. Key-frame animations are created by animators using any technique available to them in the modeling package. These animations are then exported using a custom exporter; the exporter samples each bone at the animation’s native sampling rate, generally 30Hz to 60Hz, dumping the complete set of all possible keys to the intermediate animation export file. Animators use the export tools to specify whether bones are included or not (allowing them complete control over partial, sub-skeleton animation), as well as the blending function to use, either lerp or additive.

During asset conversion (when the intermediate file is baked into a load-in-place binary resource), the animation is compressed using a lossy delta-compression algorithm which discards intermediate keyframes using tolerances that are specified by the animator in the engine’s animation tool. After this step, the raw floating-point key frame values are quantized and forced into approximately half-the space using assumptions on the ranges and integer bit packaging.

## Motion Extraction

Key-frame animations export the transformations of the reference frame of the skeleton as its own, specially marked node, so that it can be preserved throughout the pipeline and used in the game for the special purpose of “motion extraction.” This process involves using the translation and rotation from the animation itself to drive the motion of the entities in the game, rather than hard-coding such motion. The result is more believable and ultimately more flexible. “Slipping” is avoided as the animated entity is moving exactly as the animator had intended.

## Event Firing

Animators and designers both can attach event tags at specific keyframes using the animation editor. Game code can attach “event listener” objects to be notified when an event occurs (e.g., left foot, right foot, impact, liftoff, etc). Additionally, scripts are automatically hooked up as event listeners if the script exports the proper function for receiving callbacks. Together, these two mechanisms for receiving notification at specific keyframes allows for running context-specific logic at very precise moments within an animation.

# Procedural Animations

Procedural animations can also be implemented in the animation system as derived animation track types. Currently, we employ procedural animation tracks to control reference frame motion for entity orientation, and this could be extended to full IK or Ragdoll. Additionally, the game makes use of explicit blending based on game code and scripts. Such blending is achieved using a type of derived animation track that allows clients to add any number of sub animation tracks. Each sub-track is assigned its own blend strength value (in 0 to 1); this value is simply updated as necessary from game code and/or script. An example of such a case is using the gamepad to control the direction of aiming or firing; using the gamepad stick values, the resulting animation state is seamlessly derived by blending between four cardinal animation poses (left, right, up, down).

Scripting

# Overview

Scripting is supported in-game using the open-source Squirrel scripting language and runtime. It provides a syntax similar to that of JavaScript, with table-style implementation similar to and based on Lua. It is a full-featured scripting language with support for classes and inheritance, dynamic typing, delegation, co-routines, and automatic memory management (mixture of reference counting and garbage collection). More information about Squirrel is available at its homepage, <http://squirrel-lang.org>.

Squirrel was selected over Lua for several reasons, including its reference-counted memory management that obviates the need for garbage collection in many cases, native support for classes, ease of C++ integration, and a syntax that more closely matched what script implementers were comfortable with.

# Binding Scripts to Entities

Scripts are managed in-game using the Script Entity. Script Entities attach to other entities through the parent/child relationship; the script code itself acts upon the entity to which the script (and hence the Script Entity) are bound.

From the designer’s standpoint, scripts are simply assigned to entities in the scene graph editor SigEd. Any independent entity in the SigEd file can have a script assigned to it; additionally, a script can be assigned to an entire file, and sub-entities in the file can be accessed in the script using names and/or tags.

# Exposing Native Code to Script and Vice Versa

Both native C functions and C++ class types can be exposed to Squirrel using its binding facilities. The game engine has added functionality to this process that enables associating documentation of the exposed classes and functions for scripters to use. Examples of exposed functionality include basic math structures (vectors, transformations), entity creation and manipulation, animation control, audio and sound playback, as well as hooks into all game-specific entity types (soldiers, artillery, projectiles, etc.).

Additionally, game code may expect scripts to have implemented specific functions to be called for various events. These script functions are called from C++ native code via our custom script function exporting mechanism (see below).

# Game Engine Extensions

In addition to the built-in Squirrel language facilities, the game engine has added a few extensions and keywords that are detected during the offline pre-process and compilation to byte-code. These extra features enable scripts to declare explicit resource dependencies (see the Resource system for a description of sub-resources), as well as to declare “exported” functions that the game code is expecting to find.

Exported function names are mangled in such a way as to ensure their uniqueness across all loaded scripts. E.g., if game code is expecting to find a function by the name OnSpawn, then any script wishing to implement this function will need to use the special syntax:

sigexport function OnSpawn( … )

{

}

The preprocessor will find the ‘sigexport’ tag and mangle the actual function name, while keeping it associated with the user-friendly name “OnSpawn”. A set of expected export functions are bound together using the idea of script classes (the word “class” here is used in the general sense, not in the C++ sense). E.g.:

sigclass Soldier

sigexport function OnSpawn( … )

{

}

sigexport function OnHit( … )

{

}

// etc.

Game code can define script classes which are bundled with a set of expected export functions; together these form script function tables which encapsulate the details of invoking these script functions in native C++ code. They also allow us to think of individual scripts as an entire “entity definition”.

Scripts are not required to use this ‘sigclass’ convention; scripters are free to share functionality in regular scripts and invoke this functionality from the entity-specific scripts which export a particular interface.

AI

# Overview

AI in the game encompasses all units in the game that must make decisions. Much of the time, this decision-making entails figuring out where to go (pathfinding), but can also entail targeting, spawning decisions, or other runtime selection.

While there is no one sweeping “AI” sub system in the code, we describe it in broad terms as the mechanism by which entities are controlled, and hence caused to animate and move. This control is implemented primarily in script and game code.

# Behaviors

Behaviors consist of a mix of game code and script. Game code is implemented in such a way as to provide maximal parameterization from script, so that we can create variety and customization with data. In this sense, the system is data-driven; code-reuse is emphasized using this simple tenet: concepts exist in code, and specifics reside in script. A simple example of this mantra: “tankness” is defined in code through a set of classes and functions, defining steering and projectile launching; but the specific parameters, and potentially a small amount of behavior, all reside in code.

By staying data-driven on parameters and simple logic, designers can quickly iterate and make changes without requiring a programmer for every single change. On the other hand, by keeping the core of the behavior logic in native C++, we insulate designers from having to deal with the challenges of software engineering.

Behaviors are implemented using a simple state system. Agents can be considered to be in one state at a time. The behavior logic associated with a given state can potentially be separated between both code and script. Agents can be interrupted from their current state by reacting to events (implementing event handlers); alternatively, the current state itself can elect to change to a new state.

# Pathfinding

Pathfinding is accomplished using two core techniques: waypoint/path following, dynamic obstacle avoidance.

## Waypoint/Path Following

Designers can place waypoints on the level maps, along with connectivity and bounding meta data, providing the basis of the primary route planning for agents. Agents will evaluate their route periodically to assess which route is preferred. The waypoints will broadcast dynamic information at runtime to the agents, allowing agents to change their route or prefer one route over another depending on circumstances (i.e., if the player is concentrating on defending one area very heavily, then the units may cease to want to go that way, unless it’s their only option, etc. – similarly, if one part of the map is destroyed, as in a bridge, then units will be able to select alternative paths).

## Dynamic Obstacle Avoidance

In addition to following their primary route, agents will be faced with numerous static and dynamic obstacles on the way. Agents will perform ray casts with their immediate environment to determine potential blockers en route to their destination, and will adjust their paths to steer toward the edge of their blockers’ bounding volumes.

Additionally, for visual flair as well as to aid in the agents’ ability to traverse the complex environment, the engine employs a system for contextual animations (often referred to as “smart” animations), allowing the environment to inform the agent of a specific animation to execute, given that the agent has met certain criteria. For example, fences can have context animations embedded which inform soldiers to hop them when they are close enough; walls can contain context animations for climbing; barbed wire can contain context animations for crawling. When combined with pure procedural dynamic obstacle avoidance, contextual animations are a powerful force for improving the appearance of the agents’ intelligence, while simultaneously simplifying the problem of pathfinding.

It is worth noting that dynamic obstacles and context animations do not figure explicitly into the core pathfinding algorithm; they are considered only in the context of their immediate surroundings, and are dealt with as a temporary interruption to the pursuit of their primary path. As such, certain guarantees of ensuring potential paths between dynamic obstacles are imposed on the designers; in practice these limitations are minimal, as buffers are added to dynamic obstacles (units) ensuring enough room between objects for soldiers to navigate.

Destructible Environments

# Overview

The elements involved in creating destructible environments for the game include:

* Mesh Swaps, progressing from undamaged to more damaged
* Mesh Debris Pieces, driven by procedural dynamics
* Particle Effects, for smoke, fire, dirt, and other small debris
* Decals for scorch marks and other dynamic surface damage

# Creation Process

The scene graph file and mesh formats directly support mesh swaps and spawned debris pieces, such that artists can build a complete chain in a single file, supplying semantic information on each mesh so that the engine knows what is considered the primary mesh and what is considered debris, as well as which debris pieces are spawned between which mesh swaps. This process greatly facilitates authoring destructible meshes, from both a programming and art content perspective.

# Triggering Destruction

Triggering a mesh “transition” can be done from game code or script on any detectable event (e.g., hit points drop below some threshold, getting hit by an explosion, etc.); this transition causes the primary mesh to advance to its next damage state, and any debris pieces to be spawned with dynamics (with parameterized settings, such as force). To add to the overall experience, particle effects and decals can also be spawned from the same event, either in script or in game code, using straightforward API calls.

# Physics

While the feeling of physically realistic dynamics will need to be conveyed, the actual toy-nature of the game, combined with the amount of debris, allows for a simplified physics model which we used in Toy Soldiers 1, using internal physics and dynamics code. This physics handling is largely built to take advantage of known circumstances and assumptions, allowing us to write less-CPU intensive routines, as well as to specifically create an exaggerated, over-the-top, “toy” feeling throughout. We will be building off this system for Toy Soldiers 2, and do not intend to use an external physics SDK.

Networking

# Note About Scope

As it stands at this time, the scope of the game itself is still open pending approval of additional programming resources allowing us to increase our focus on improving the network infrastructure in TS2. In the initial, pared-down design, TS2 would support two player head-to-head in the same vein as TS1. However, in the ideal case, allocating a dedicated network programmer opens up the discussion to 4 player multi-player – this would include multiple modes (head-to-head, co-op, others), and necessitate more complex match-making, lobby support, and host migration, not to mention a more complex synchronization technique – all features that would without a doubt improve upon the TS1 multi-player experience. For the continuation of this section, we will assume the addition of an additional programmer – however there is a section on fallback strategy at the end.

# Synchronization

The underlying architectural mechanism for keeping connected players’ simulations synchronized is one of the most critical decisions affecting the entire online aspect of the game – it affects the allocation of resources for both development and test, it affects how gameplay code is written, and it affects the way in which end users will interact with and enjoy the multi-player experience over LIVE.

With the weight of this decision in mind, we break it down in terms of 3 high-level, fundamentally different approaches:

# Lockstep

Under a Lockstep model, connected peers send only the players’ inputs – all synchronization relies upon deterministic simulations which, given identical initial state, will play out identically no matter the length of the game or events that transpire during gameplay.

## Pros:

* Minimal “networking code” – the only data actually transmitted is the gamepad or similar representation of input.
* Can be treated in many respects as equivalent to a local multiplayer game, simplifying many aspects of gameplay code.

## Cons:

* Requires extensive testing to ensure robust synchronization.
* Requires extensive log-based debugging to identify and fix out-of-sync conditions.
* Requires strict separation of gameplay code into objective vs. subjective state – and never the two shall meet!
* Fragile, “butterfly effect” problems – one unsuspecting, minor change in gameplay code can result in out-of-sync conditions later in seemingly unrelated code.
* Little flexibility/tolerance for bad connections or high latency, resulting in the perception by players that they are often playing laggy games.

# Distributed Entity Ownership

With Distributed Entity Ownership, the state of all the entities in the game world is “distributed” amongst the connected peers. “Real” state data is transmitted between the peers under varying schemes in an effort to minimize data transmission, while keeping everyone in sync. The conditions for ownership can vary, with the primary goal being that the entities a client “cares most about” are owned by him or her so that they can all be simulated locally – however this brings up complications when multiple clients “care a lot” about the same entities.

## Pros:

* Flexible model for data transmission that can better tolerate lag and poor internet conditions.
* Accommodates client-side prediction/simulation in between authoritative state sync.

## Cons:

* Ambiguous modes of handling entity ownership conflicts: i.e., resolving any physics-based interactions can be difficult if the objects involved are owned by separate peers.
* Increased data transmission requirements over Lockstep.

# Client/Server

Under Client/Server, one of the connected players’ consoles is selected to host the authoritative simulation of the game world. Clients send their input to the server, and receive authoritative state back from the server. In the meantime, clients employ prediction and non-authoritative simulation to present an uninterrupted view of the game world for the client.

## Pros:

* Flexible model for data transmission that can better tolerate lag and poor internet conditions.
* Accommodates client-side prediction/simulation in between authoritative state sync.
* Well-understood paradigm with many resources/materials to draw from.
* Little to no ambiguity over entity ownership.

## Cons:

* Increased data transmission requirements over Lockstep.

# And the winner is…

In our estimation, the best solution for TS2 is a Client/Server approach. By minimizing, distributing, and lowering the frequency of state updates, data transmission can be tamed sufficiently to allow for minimum spec bandwidth requirements. To compensate and hide any latency, low-frequency updating, the game will rely heavily on client-side prediction and simulation to bridge the gap between authoritative updates from the server.

Client/server is a tried-and-true approach for many FPS-style games (if not *all*), and we believe it is the best fit as we move the Toy Soldiers franchise firmly into the action/shooter space. In the worst case (i.e., if we are breaking our bandwidth constraints), unit counts can be reduced for online modes, relying on custom maps that cater more to other aspects of gameplay. Client/server will be the cleanest way of scaling up beyond 2 players, simplifying the process of host migration. Additionally, it will provide authoritative entity conflict resolution and unambiguous entity ownership, avoiding many problems that could come up in a Distributed Ownership model.

# Potential Networking Libraries

Toy Soldiers 1 was completed using QNet and Lockstep networking – we were not satisfied with either approach, even though they made sense at the time for TS 1. Given our experience with these options, we have concluded that an ideal approach would avoid both – however the degree to which we are able to seriously consider new approaches depends on increasing our programming resources.

To start with, for the underlying networking libraries, we will investigate using the XSESSION APIs directly for all session/matchmaking related functionality. Initially, our thinking is that we would prefer operating more closely to the actual lowest level API in order to manage our reaction to network state changes as rapidly as possible – in our experience, relying on QNet meant dealing with a black box that was often unpredictable and unresponsive.

On top of the XSESSION APIs, we will need a transport protocol – we will investigate both XRNM and ENet as two potential channel-based point-to-point messaging APIs. XRNM is provided as part of the XDK, and ENet is a freely licensed open source solution. Both of these solutions will provide greater flexibility than QNet in controlling the delivery of different message types, particularly if they have different requirements (reliable, unreliable, connected, etc).

Another option under consideration is using a commercial networking library such as Quazal’s Net-Z – theoretically, such an option could provide solutions for all areas associated with networking, including matchmaking, sessions, and data transport itself.

# Fallback Strategy

In the absence of increased programming resources, we intend to maintain the same level of multiplayer features as was present in TS1 – notably 1v1 head-to-head gameplay over LIVE. Any additional resources would be devoted to improving the overall integrity of the core experience using QNet and Lockstep, the solutions that were used for TS1. A greater investment would be made up front in developing sync-verification measures for Lockstep, and more care would be given to the various edge cases associated with QNet, including game invites, session joining, matchmaking, and potential disconnect and error messaging scenarios.

Saved Games/Serialization

# Overview

For TS1, the requirements for saved user state were minimal. Players were only able to save between maps, meaning no “in-battle” or generalized world-state needed to be serialized; only the player’s current progress in the campaign-mode needed to be saved, along with various statistics. Given these requirements, we were able to get away with storing saved games in a small, simple data structure that was serialized directly to/from the profile memory space.

TS2 will require a more sophisticated approach to serialization that is able to support multiple scenarios:

* Checkpoint-based save games in Campaign Mode play
* Join in Progress in LIVE Multiplayer
* Incremental authoritative state updates under Client/Server
* Debug facilities (robust network sync checks)

These scenarios require the ability to serialize the entire world state, and to be able to store it and potentially send it in a compact form. The serialization system will need to be flexible enough to accommodate these scenarios, as well as others that may come up. Hence the system will be built from the start with the idea of supporting options, so that entities can save their state differently depending on the context. Additionally, the ability to compress data both during serialization (i.e., an object that has knowledge about the range of its data can write fewer bits to the save-stream), as well as post-serialization (operating on the raw, opaque bit-stream after all objects have written to it) will be supported.

Because the storage requirements for TS2 will likely exceed the size of the per-title profile storage, we will also have to add support for multiple storage devices, rather than relying solely on the simplified profile API.

Memory Management

# Overview

Memory is managed using various allocation strategies depending on the context. At the coarsest level, memory is managed independently (using isolated heaps) for Main RAM and Video RAM. With regards to this separation, it is worth noting that despite the unified memory model of the Xbox360, one cannot simply allocate textures and geometry using the same memory that one could use for standard data structures, due to restrictions imposed by the hardware (physical, un-cached memory requirements for data used on the GPU, which would lead to terrible performance characteristics for data used on the CPU). Nonetheless, the fact that the game can specify at startup overall heap sizes between Main RAM and Video RAM is very helpful.

# Main RAM Allocators

Main RAM allocation strategies separate the problem into two primary categories: very large, infrequent allocations versus frequent, “smaller” allocations. To service the large, infrequent requests, we employ a straight-forward free list of available raw memory chunks, whereas for the smaller allocations we employ a fixed-size pool-based strategy.

## Large-Buffer Allocation

Large buffer allocation prefixes a header at the start of all outgoing allocations, allowing the allocated memory to be tracked in the “free” and “used” lists. When a new request comes in, the free list is searched for a piece that is big enough to service the request; if an exact fit is found, the memory is moved to the used list, and a pointer to the user-available memory region is returned. If a larger piece is found, then the memory left over is split into its own chunk (with its own header) and added to the free list; the existing chunk’s user-memory size is updated to reflect the request, and then moved to the used list.

## Memory Pools

We place more of an emphasis on solving the frequent, “smaller” allocations than on the larger, infrequent allocations, as their impact on the performance and fragmentation of the game are much greater. To this end, we employ fixed size allocator memory pools in two different contexts.

The first case in which we use memory pools is as a layer on top of “anonymous” allocations using the global new operator. Allocation sizes are checked against a global list of pools, and the pool with the smallest allocation size which is greater than or equal to the requested size will return the memory.

Additionally, memory pools can be bound to specific classes by using the class new and delete operators. These pools are slightly more efficient, as there is no need to check against multiple pools, and the fixed pool size will be exactly that of the class instances (with some adjustment potential for padding/alignment). Only class types which require frequent dynamic allocation will receive these special “class-specific” pool allocators (usually dynamic entity types).

# Video RAM Allocator

The Video RAM allocator follows the same general scheme as the Main RAM large buffer allocator. However, headers need to be stored in Main RAM, and so cannot simply be prefixed to the user memory. Headers are tracked identically, but simply point to the user memory region to which they correspond.

Additionally, the Video RAM allocator needs to adhere to the various requirements of XMemAlloc, as defined in the XDK.

Multi-Threading

# Overview

Several tasks in the game lend themselves to an asynchronous approach which will aid in distributing them across the available cores and hardware threads the Xbox 360 offers. The initial set of tasks that will be considered for multi-threading include file loading and decompression, ray casting and spatial queries, particle system processing, animation evaluation, and audio streaming.

We intend to employ two styles of multi-threading. For systems which require potentially lengthy atomic operations that can span beyond the duration of multiple frames or that require constant and immediate attention, we intend to dedicate an entire thread. For other systems whose workloads might vary significantly from frame to frame, but whose individual tasks are relatively short, we intend to use a distributed job system where tasks can be allocated to varying threads.

# Rendering

The process of rendering involves several distinct operations which deserve multi-threading attention, albeit with specific strategies designed for each task.

The first step, broadly, being spatial database queries, will be handled initially by separating each query into a job, and then running all queries simultaneously on separate cores. There are multiple queries required for retrieving lights, shadow casters, and rendered objects; additionally, multiple viewports will require extra queries.

Once the required object and display lists have been generated, these need to be sorted by state and drawn. While individual list sorting can be easily solved using jobs, the actual draw call issuing will require special handling as the device can only be used by on thread at a time. To solve this issue, we will investigate the generation of pre-compiled command buffers representing batches of draw calls and state application on individual threads, before re-playing these command buffers once generated on the primary device ownership thread.

# File Loading and Decompression

File loading and decompression is discussed in more detail in the Resource System section. With regards to multi-threading, it is worth noting that this process will likely receive a dedicated thread, due to the high priority nature of getting files loaded into RAM as quickly as possible, as well as the more complex interplay between the actual hardware I/O and software decompression. However, due to the lack of a significant streaming content need, the majority of resources should be loaded with the map, meaning this thread may be used as an auxiliary thread for job requests if the incoming file load is non-existent.

# Ray Casting and Spatial Queries

Due to the dynamic, 3D nature of the play field, units need to frequently update their view of “what’s around them” (call it their “proximity”). This process involves various spatial queries and ray casts, which, when multiplied over hundreds of units, can become very expensive. Fortunately, the units rarely need to update their proximity every frame. As such, this process is easily adapted to an asynchronous job system where units push requests for updates into a lock-free style job queue, and at some later time receive the results to their updated proximity query.

# Particle System Processing

Due to the high number of particles to be deployed as part of the explosive and destructible environment, the processing burden can grow quite high. Distributing the particle load across the cores should aid our frame rate. Particles are particularly amenable to the job-system style of multi-threading, due their primarily one-way style of interaction: spawn a particle system with certain inputs, and forget about it. Particle systems may need to poll for their inputs each frame, but in general, no system other than rendering is reliant on the particle systems, meaning they can easily be deployed in an asynchronous fashion.

# Animation Evaluation

Animation Evaluation in many ways follows a similar pattern to ray casting and particle system updates, in that “requests” for evaluation can be pushed to a job queue fairly simply. However, the output of the animation evaluation is often needed shortly after the evaluation itself, generally in the same frame, and certainly before rendering.

To facilitate this process, animation evaluation has been separated into two distinct parts: reference frame evaluation, and skeleton evaluation. So that the motion of the animated entity can be known immediately, we can evaluate just the reference frame of the skeleton, while delaying the evaluation of the rest of the skeleton. This separation allows us to know immediately where an entity’s movement will place it in the world, so that we can continue with other game code and scripting that relies on this knowledge.

The more difficult task lies in delaying the processing of events that still rely on the animated state of the rest of the skeleton. These events are generally reliant on an attachment point which is bound to one of the bones. The available options are to queue up these events and execute them toward the end of the frame after all animation is performed, or else to delay them a frame to prevent stalling. Delaying a frame will introduce small graphical inconsistencies. We will evaluate which option to pursue based on the severity of the graphical inconsistencies and the time spent trying to avoid them.

# Audio Streaming

Due to the constant and time-sensitive needs of audio streaming, it will receive its own dedicated thread. This thread will be managed as part of the WWISE library.

Localization

# Specifying Languages/Locales

In order to minimize the need to recompile the game executable for adding a language, the list of supported languages will be kept in an external script. This script will simply enumerate the languages, providing the game with metadata relevant to each language (the name, language-specific global resources/fonts, etc.).

# Localization of Text

All text that is intended for localized display will be stored in separate, individual xml files by language (using Unicode text formatting). Files will be named using the language they represent (e.g., English.xml, French.xml, etc.). The format of these files will be very simple, essentially containing a list of entries whose tag names represent the unique identifier of the text. For example, in the English language file:

<A0914>Hello!</A0914>

And in the French language file:

<A0914>Bonjour!</A0914>

These text entries are linked by the fact that their tag is identical (in this example A0914). These xml files will be treated as resources that can be referenced by game code or script using the tag ID of the text block (e.g., ShowMessage(“A0914”) would display a message yielding the text “Hello!” in English, “Bonjour!” in French, etc.). Retrieving the Unicode string at runtime is just a simple matter of opening the appropriate language’s Unicode text file resource, finding the entry with the specified tag (a single-byte-char string, non-Unicode, used to uniquely identify the text), and then returning the related Unicode string for display. Once the actual Unicode string intended for display has been acquired, it can be rendered using the proper language bitmap font pack.

## Dynamic Strings

Clearly, not all text can be statically determined before the game runs. To provide for dynamic localized text, we will support formatted display of script and entity variables from within text blocks in the localization text files. By using a special syntax and character escape sequence, designers (and localizers) can combine language-specific text with dynamic numbers and strings from the game.

# Localization of Images

While we intend to minimize the need to localize images, we will provide for the need using a fairly simple file naming convention. First, we will assume a default image located at whatever path is specified (e.g., “hud\warning.tga”). Before loading the image, we will look for a language-specific override file by looking in a folder of the same name as the specified language (e.g., “hud\lang\_french\warning.tga”). This convention will allow for flexible modification of images requiring localization without any modification to code or script.

Asset Pipeline

# Overview

The asset pipeline includes content creation tools, the “intermediate” files they produce and edit, the asset converter, and the baked files it generates to be used in-game.

|  |  |
| --- | --- |
| Asset Pipeline | Intermediate files are generally stored in xml format, though that is not an explicit requirement. These files are designed to be easily extended and modified as the tools gain functionality. They are also designed to be easily serialized. While concern is paid to efficiency, this is not the primary concern for the intermediate files.  On the other hand, baked files are designed to load and be used in the most efficient form. The Asset Converter is responsible for taking the intermediate file in xml format and “cooking” it into the more efficient binary form that is consumed by the game. Automation The conversion of all intermediate assets is easily automated via build scripts to perform clean rebuilds, as well as incremental updates of only assets that have changed. Additionally, pack file generation is also easily integrated into the process using build scripts.  The generic build scripts operate on the entirety of the “Res” depot, but our internal tools allow artists and designers to easily rebuild individual assets and their dependencies. |

# Source Assets

Raw-form source assets from which our intermediate files are exported are included in the main resource directory tree. These are kept in folders that start with a ‘~’, generally named “~src”. The asset converter skips these folders, as they are not directly converted to game assets.

External Technologies

# Listing

The following external technologies have been discussed in other sections, but are listed here for completeness. Note that some of the technologies may have been discussed in the context of potential use or evaluation only, and as such are not guaranteed to be used in the project; these items are marked as optional in the list. Additionally, any technologies that are part of the XDK are not included.

* WWISE Music and Sound Effects
* Squirrel Scripting Language
* zlib 1.2.3 Compression Library
* ENet: UDP Networking Library (optional)

Coding Standards

# Overview

In general, the coding style used is object-oriented C++. Global objects (and hence singletons) are avoided, and where used should be confined to file-scope. Methods requiring use of a global object should prefer including the object as a parameter to the method, in order to make dependencies explicit and refactoring easier.

Objects that allocate and deallocate memory internally are responsible for properly implementing copy semantics to prevent leaks. Additionally, naked pointers are strongly discouraged: nearly any context requiring pointers can be made to use smart pointers without any performance loss.

While use of standard libraries (e.g., STL) is not explicitly banned, it is generally discouraged; custom solutions to many popular containers are implemented as part of the game engine, and their use should be preferred for reasons of performance, standardization, memory-usage, and compatibility with our load-in-place serialization.

# Notation Conventions

In general, we attempt to avoid draconian “Hungarian-style” prefixes and/or post-fixes. However, we do employ a simple, single lowercase letter prefix convention throughout the code. The following are the single-letter prefixes used:

|  |  |  |
| --- | --- | --- |
| **Prefix** | **Example** | **Description** |
| t | tSomeClass | A prefix for the name of a type (class, struct, enum, typedef, etc.) |
| f | fSomeFunction | A prefix for the name of a function or method |
| m | mSomeVariable | A prefix for the name of a member variable |
| g | gSomeVariable | A prefix for the name of a global variable |
| c | cSomeConstant | A prefix for the name of a constant variable |

The over-arching reason that we employ any prefix-based convention at all is to avoid name clashes. For example:

class tSomeClass { };

class tAnotherClass

{

tSomeClass mSomeClass;

public:

tSomeClass& fSomeClass( ) { return mSomeClass; }

};

By following this convention, we cleanly avoid any clashes between the type, the variable, and the function exposing the variable.

One additional side-effect of this convention is that it tends to provide a unique signature to code written in the game and engine, making it easy to distinguish between game code and external library code. It is also a simple enough scheme that a programmer doesn’t need to keep a cheat sheet to remember.

Risks

# Overview

With TS1 completed, we have a good sense of the biggest risks affecting the project.

# Networking

Networking is the primary area of risk for TS2 from a technical standpoint – see the Networking section for a thorough discussion.

# Performance

As Toy Soldiers gameplay revolves around pushing as much mayhem, dynamic destruction, particles, and overall unit counts as much as possible, performance will always be a risk – that said, we feel we have created a robust multi-threading architecture that will provide the basis for addressing these performance bottlenecks. However, it will require constant attention, profiling, and content adjustments to ensure we hit our performance targets.