|  |
| --- |
| Author: Jeff Lansing |
| Post Mortem: GXT |
| The Generic XNA Toolset |

|  |
| --- |
|  |

***Introduction***

For the past couple of months I’ve been working tirelessly on a game engine built on top of the XNA Framework. After getting the project registered as an Independent Study, development has accelerated considerably, and with the quarter coming to close it is time to wrap up the core components of the Generic XNA Toolset (GXT). Although the independent study is almost over, this does not spell the end for the project. Given that I have divested enormous amounts of time into it already, I am very determined to release a mature version of the project on CodePlex sometime over the summer. This document is intended to help the reader understand how the architecture works, the design decisions I made, and what I learned from the project.

There were a number of motivations for completing this project. Game engines are a specific area of interest for me and I wished to explore the field further. While reading on how they work is important, nothing is a valid substitute for writing your own implementation. In doing so you are able to truly understand how components are intended to work and are able to prove this to others, most notably, employers. It is my hope that GXT will help me build a portfolio of my capabilities. It is also possible that I will use the framework with a team to produce a finished game.

I will likely create a user guide as a separate document which will contain far more code examples. In contrast, this document is just intended to serve as an abstract explanation of major components in the library and provide proof of the work I put in for the independent study. If you are looking for more specifics you can look at the *Doxygen* generated documentation for the project online (<https://people.rit.edu/~jrl6166/gxt/namespaces.html>).

***Features Overview***

A Snapshot of the Major Components Implemented in GXT

*AI*

AI solutions are typically very application specific, and therefore making a general purpose AI solution is a bit silly. However, path finding is a very commonly needed solution, so GXT supports that on the A.I. end, and only that. The A\* algorithm is utilized to find efficient paths. A broad phase collider for the path nodes is pending that will further speed up searching. I’m not sure yet if I want to provide state machine support or leave that up to developers extending the framework.

Robust collision queries are not technically just a feature of an A.I. system, but are still usable for a good A.I. system. This is one of the many benefits to using collision filters and attaching object tags to in game geometry.

*Animation*

Since the scene nodes manage parent-child hierarchies perfectly, I decided to simply extend this model into my animation system. The animation system lets the user define keyframes, poses at these frames, and full clips with support for time scaling, things like looping, and playing the routine backwards. A controller which performs operations on string aliased clips is built on top of this to provide a nice interface for the developer. While this may sound complicated, it is really just a lot of relatively grungy code built on top of a linear interpolation for the position, rotation, and scale of the pose. A robust animation tool, which will be a requirement for artists, is planned for future development.

*Audio*

General purpose audio solutions were also a bit difficult, but mostly due to the way XNA is structured. XACT is not the most robust piece of software ever created, and the flexibility you have with it is project specific, since most definitions are customized by the sound artist. Initially I wanted to be able to perform sound transforms on hierarchies of grouped sound effects, but found that default XACT files could only do this for volume. I might later require the use of project files which included extended definitions, such as pitch or tempo. Sounds can be driven by a controller whose interface is similar to the ones used to play animations. Microphone support is currently unimplemented but pending for the future.

*Collision*

GXT has full support for Axis Aligned Bounding Boxes, Oriented Bounding Boxes, Spheres, Rays, and Convex Counter Clockwise Polygons. Tests can be performed between different pieces of geometry, and there is support for numerous different queries (e.g. distance, closest point). Generic interfaces are provided for broad and narrow phase colliders which heavily optimize the speed of batched routines. Most parts of GXT will use the Sort and Sweep algorithm for the broad phase and GJK for narrow phase computations. The Expanding Polytope Algorithm (EPA) is used in conjunction with GJK to process collision information (depth, normal, contact points). All in all, the collision detection in GXT is probably the most mature and feature rich component of the project. As a result, there aren’t any major plans for future changes, although I would like to write a Dynamic Bounding Volume Tree and support other shapes (like Circles and polygons with curved surfaces) in for narrow phase collisions.

*Physics*

Implementing realistic physics was perhaps the most time consuming and challenging part of the project. GXT supports a system of convex polygons, dynamic rigid bodies, world cast queries, and a solver for object interpenetrations. Geoms support multiple variables for collision groups, flags for collision response, and events which can be fired in the event of a collision. Rigid bodies can have forces, torques, and all of its internal attributes set manually by the user. Rigid bodies also support multiple motion types akin to the system used in *Havok Physics*, so not everything needs to be controlled by forces. Such decisions eased the integration of dynamic bodies and keyframed animation. The solver is currently a bit buggy and improved model is pending. Joints and other constraints are planned for the future as well. There are no plans at the moment to support swept tests or a robust TOI solver.

*Processes*

Over the Summer I read through the book *Game Coding Complete* and really liked the “process” architecture model it presented. As opposed to a simple update method and enabled flag, it was much more flexible at initialization, identifying process types, managing states, and disposing of itself when it was finished. The key component of the gxtProcess structure that differentiated it from other models I had created in the past was the ability to chain processes. Each process has a next process that is run when the host process is killed. This allows you to create a sequence without an overarching structure that keeps track of states. Since I’m trying to stay focused on general purpose, the only included processes include wait processes and timers, but I’m looking for more meaningful ways to use them in my engine. Regardless, most processes will be application specific (e.g. SpawnPlayer, EndLevel, etc. etc.).

*Controllers*

A very light controller interface is provided and easily added to the main gxtWorld class. Throughout the project I was determined to keep things decoupled and inheritance trees as shallow as possible. Overriding a camera for a new tracking behavior isn’t really best understood as an entirely new camera. By sticking with a controller interface I kept inheritance down to 3 levels max across the project, improved encapsulation, and in general insisted on a better design practice. An animation interface that facilitates the playing, stopping, and rewinding of string named clips is implemented as a controller.

*Actors*

Actors, like controllers, are simply implemented as one interface in the engine. Although this is light (again in the hopes of being general purpose), it doesn’t mean actors aren’t important. Intended to be placed in the game world, actors have simple methods and properties which handle the loading and unloading of their attached resources. Most notably, each actor has an UnloadActor function that provides one function to dispose of every associated resource.

Actors also feature Hashed String Ids for optimization purposes. According to books like *Game Engine Architecture*, string comparisons, particularly in long if-else ladders, can be an enormous bottleneck. Since integer comparisons are faster, I made one class that wrapped an int and a string, and operated on a variable hash function which could be defined at startup. Very involved functions like MD5 were too expensive for runtime, but some simple ones, like the Jenkins One At A Time Function offered reasonable disparity and excellent speed.

*Input*

GXT supports managers and object instances for keyboards, gamepads, and mice. The singleton managers handle updates for all devices, which track their current and previous input snapshots. Event handlers for connection/disconnection events are implemented where XNA supports them. The system is also built on a very useful control state concept (DOWN, FIRST\_PRESSED, UP, FIRST\_RELEASED) which is very useful for processing input queries. This support is extended to macro button combos as well. I’m currently quite pleased with the input system I’ve layered on top of XNA, but I would like to come up with a cleaner way to process character entries on the keyboard, support repeat rates, and come up with a generic interface for control schemes.

*Rendering*

The rendering system is based on parent-child transform nodes that can operate with attached drawable objects. After careful implementation, I was able to get a solid system which supported the inheritance of position, rotation, scale, color, and visibility. I also put flags in each node that made this inheritance optional (e.g. inherit position but not scale).

Other components of the rendering system were just as vital. For example, display management properties had to be handled properly throughout the system which proposed some unique challenges. I wanted to define a camera with a position, rotation, and zoom parameters, and simply pass the matrix to SpriteBatch to process transforms on the entire scene graph. However, when I changed resolutions results would be skewed. This required applying screen scaling and registering cameras to the display manager to be notified when the resolution was changed. There are no plans to produce any kind of 2D lighting schemes or particle systems. The time needed to produce such components would be too great to justify the time taken away from other, much more pressing concerns.

*Debug/Log*

A lot of tools are provided to ease things on the developer end. A log system is implemented in place of basic print statements. Log entries can be made in one place and processed by any number of attached log listeners. The log system also supports variable levels of verbosity, can tag listeners for removal, and update them once a frame. This last bit was particularly useful to delay expensive buffer flushing for log files. Currently the log system can produce color coded output to the system console, into an html file, and into a pending in game console.

Core debug utilities include functions to retrieve the current fps and memory usage, as well as a custom assertion method built to work with the logging system. A general purpose in game console is in development that will hopefully exploit C#’s Func class and Linq to attach any kind of action to a command.

The most useful debug component is probably the debug drawing queue. With simple procedural calls, you can add lines, points, AABBs, polygons, and a whole lot more in any color at any position in the world. This makes debugging much easier because it allows the developer to actually visualize the values in game rather than from a log file. Such a system is a must for physics engine development, and is also useful for placing ugly textures in the place of ones that were not found or loaded properly.

*IO*

The total amount of IO development for the project was relatively light. Most of the imperative to provide my own classes at all was to get platform independent wrappers that could work on the PC and the Xbox 360. Right now the system does a nice job reading and writing INI and XML files. I didn’t see much of a point in creating a resource manager or compile time types until I was certain how my runtime objects would be structured. This is still very important and one of the last major components I wish to complete for the project before it sees its first release.

***How GXT Works***

*Startup*

GXT’s startup routine is heavily inspired by the methods employed in Ogre3D. One singleton manager called gxtRoot handles the management of all the major “low-level” subsystems used in the engine. Although XNA’s given framework makes early startup easy to understand its “GameComponent” model fails to differentiate low level components from game worlds and logic. gxtRoot is designed to handle the management of all components usable in all game screens, and not just serve as a kitchen sink for everything from game worlds to log systems. There is really no reason for a graph of path nodes to be active or allocated if we are just loading the main menu.

Using the gxtINIFile reader, a custom config file is parsed, or the default startup file is used. Settings at this level define if components are enabled or should even be created in the first place. While settings may be defined for components outside of gxtRoot, only these are created at startup. Such components include managers for logging, input management, debug drawing, and other utilities that are shared across all game screens.

*Main Types*

*Typical Update Cycle*

Game

Root

Update Root Subsystems

Update Log

Update Gamepad Manager

Update Keyboard Manager

Update Mouse Manager

Update Debug Drawer

Update Audio Manager

Update Game Screens

Menu Screen

Game play Screen

Update World

Update Debug

*Typical Draw Cycle*

Game

Root

Draw All Game Screens

Game Play Screen

Draw Scene Graph From Root Node

*A Closer Look at World Updates*

Get Adjusted GameTime (dt \* game speed)

Update Actors

Manage gameplay logic, adjust values of internally referenced components as needed

Update Controllers

Modify another object based on a behavior/input

Update Animations

Convert global to local normalized time

Handle wrapping/looping

Interpolate between animation states in active tweens

Update Process Manager

Update all active processes

If uninit, run init routines

If dead, remove and replace it with next process (if any)

Update Physics

Process added/removed items

Apply world gravity to rigid bodies

Update all awake rigid bodies

Apply Euler formula

Clear forces on body

Update all geoms (with attached rigid body)

Update AABB in broadphase collider

Process Broadphase

Get collision pairs, remove old ones

Process Narrowphase

Test collision pairs

Get intersection bool, penetration depth, collision normal, and contact points

If intersecting, fire collision events, process collision response in solver

Update Audio Scene

Prune inactive cues from categorized collections

Update existing sounds/cues

***Design Decisions Made In GXT***

*Why Assertions?*

In most commercial game engines, SEH (structured exception handling) is simply too expensive, particularly for release code. As an alternative, developers use assertions, typically in the form of a macro for two reasons. One, assertions heighten the incentive to fix bugs to a much greater degree. Two, macros can be disabled with careful use of preprocessor directives, thereby effectively removing them from finished code. This is important because the parameters to an assertion can often be quite expensive.

Unfortunately, disabling exception handling in the compiler was not going to happen, and everything, absolutely everything in the .NET framework uses exceptions for error handling. This does not, however, mean assertions were a waste to develop. I made my own assert function which tied nicely into my log system, and using C#’s “Conditional Atttribute,” was able to effectively remove them from release code. Using assertions very liberally, particularly early in development is very good practice and help me find bugs much earlier than I would have otherwise.

*Why Single Threaded?*

I made a conscious decision, early in development, not to support multiple threads. There were a number of reasons I chose to do this. One, since a lot of this was a “black box” to me, I determined it would be more worthwhile to go ahead implementing new things I have never tried before rather than getting familiar code to work with threads. The shift to thread support is painful in every project, but it is easier to swallow if you design for it from the beginning. If you are certain you want to support threads, especially for a game engine, this is a consideration that needs to be taken into account every step of the way.

I still decided to take a semi-step into this direction. I’d like to consider most of GXT to be very batch friendly. I tried my best to move away from “god objects” which handle the updating of all their underlying components. Therefore, in GXT, when an Actor is updated, it doesn’t process animations, dynamics, rendering, and sounds, but merely adjusts states.

*Why Controllers?*

Some might think separate controller classes are excessive, but they are definitely worth the trouble in my opinion. They make code easier to understand, decouple the interaction from the function, and extend much nicer than a model which attempts to do both. The benefits were clearer later in development when I was building on top of existing components.

*Why So Many Wrappers?*

Early in development there were times I thought wrappers were excessive and wasteful. I highly recommend wrappers because they are easy to write and offer similarity between objects and across platforms. In addition, they are much easier to extend later in development. I think you’ll find a lot of uses for wrappers beyond platform independent IO read and write calls.

*Why Only Limited In Game Profiling/Recording Tools?*

In game profiling tools can be quite useful, particularly at conveying which parts of the game are detrimental to performance. However, the time needed to develop tools beyond simple FPS and memory usage counters didn’t seem to be worth the effort when the profiler could do it for me. If I ran the engine in a window, I could still see the performance graph being plotted in another, and could see a very detailed report at the end. Some will say that “serious” developers always take the time to write these tools but I didn’t see the benefit in making such a tool that could export statistics to an excel file when the profiler went way beyond that. I similarly didn’t see a very pressing need to program recording tools when FRAPs could handle the job just fine. These systems would be ideal but eventually you have to focus on what really matters.

***What I Learned/Advice***

*General*

I learned a great deal in a number of major fields pertinent to engine development. It’s true that before I started this project I had a conceptual understanding of how most of it worked, but that only gets you so far. Even in these areas, I learned much more about them, and found ways to implement them more efficiently. At the same time, a lot of it was a complete black box to me. The GJK and EPA algorithms for collision detection were both quite involved and very much new to me. I had never pieced together an architecture this large before, and certainly not by myself. Specifics on how scene graphs worked and were implemented were still foreign to me, especially with variable rules for different inherited properties. Animation algorithms, in game consoles, excessive use of bit shifting – the list goes on and on. This project was an enormous sandbox to learn how these things worked, I’m glad I took it on, I’m glad I insisted on writing everything myself, and I’m glad how it turned out.

*Thoughts on Scene Graphs*

Every rendering system will have a scene graph of some sort, and it is vital that they are built early in development. I talk about this at length in the Lessons Learned Section. Anyways, there are some misconceptions about scene graphs that ought to be addressed. You’ll hear that they can be used for transform hierarchies, culling, and depth sorting, and so you’ll go on your way producing a structure that attempts to do all of those things. However, a scene graph which attempts to do too much in actuality will do a subpar job at everything. A tree is not the best suited for culling and certainly not depth sorting, or storing things in a way that produces the fewest batches. A separate structure will often be better for each individual task.

As for XNA, there is little reason to write culling algorithms for a 2D game. SpriteBatch is crazy optimized and you will, in most cases, actually hurt performance by doing AABB/OBB culling on quads. It can reduce the number of batches sent in instances where there are an enormous amount of drawable objects, but batches are cheap, and in most cases, my profiler told me it was adding additional computation time. My processor and free time wept so you don’t have to.

*Don’t Get Caught Up*

Design decisions for a scene graph are very important, but it is important to move forward on it. You can ponder forever which distribution of properties between nodes and the objects attached to them is best. There is no definitive answer to this question, since it will differ on the implementation. However, the answer will become much clearer when you actually start using it, and you won’t get to that point if you get stuck in this infinite conundrum too long. There are pros and cons to everything.

*Profile Early, Profile Often*

The debugger may be the most important tool at the hands of a developer, but a profiler is a close second. Profiling tools become even more important in large projects like GXT that are intended to scale well. I found an excellent profiler for my platform here: <http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/>

The most important thing to keep in mind for effective profiling: ignore the average. This has two effective meanings:

1. Be more concerned with dips and worst case scenarios than average performance. While higher frame rates are preferable, inconsistent spikes in performance are much more noticeable
2. When utilizing the profiler, don’t use simple and average test cases. Put serious strain on your code to see how it performs and understand its limits. When pushed to the max, the slower parts of the code will be more easily exposed.

*Prioritize*

Understand your core components and dependencies and write them first. No matter how hard you try to decouple components they will eventually need to coexist peacefully. If you put off writing low level components you increase the potential that you are building off of deprecated implementations. The most glaring example of this mistake in my project was with the scene graph. Initially, I didn’t have one at all and every drawable entity contained its own properties for transformations and rendering. While this was fine for testing things like the physics engine, it was an enormous omission of commonly needed functionality.

The switch over was painful, and for all the wrong reasons. If you are going to struggle over a project it should be over new and required components, not over unforeseen project management. A large amount of time was wasted rewriting the rendering system and the dependent components (like the debug drawer and all of my samples).