**Geobox Engine Scripting**

**WARNING: this manual is somewhat outdated, it was written a very long time ago (~3 years or so).**

**Brief Preface**

You are getting ready to use a scripting system that was written in one of the more difficult scripting languages and is far from finished. I am always working to improve the system and take suggestions. In the meantime, keep in mind that there may be bugs and less than stellar interfaces sometimes. I am open to suggestions and improvements to make the system more convenient and intuitive. Also, if any errors arise that are clearly my fault, please tell me immediately.

**Don’t Be a Rebel:**

There’s a time and a place for experimental algorithms and “new, fresh” coding techniques, and this is not it. Keep it SIMPLE. Simple, simple, simple is the key. Any game will be a monstrously complex piece of software and the only way to keep it maintainable and keep code flowing is to minimize complications, which means keeping complication out of the code. Don’t get clever and try to bypass usual scripting routes with little hacks and tricks. For one thing it’s \*not\* clever because all of that is trivial to do in lua, and another thing is that you’ll most likely do something that the system was not designed for; causing subtle bugs that will likely damage the game’s integrity and harm other people's work. This is not your free-form playground. If you make mistakes, it will hurt and hinder others on the team. Be careful, keep it simple, and don’t get “smart” by trying to work around the rules.

**Assumptions:**

I do not have time to write an entire explanation of programming and the object oriented paradigm. There are many, many resources where you can learn these topics, giving tutorials far better than I could give. This scripting system uses object oriented programming and encapsulation heavily to keep code as clean and organized as possible in the game script. I assume that you have a working knowledge of programming and a reasonable knowledge of OOP (objects, classes, encapsulation, inheritance, ect). You don’t need to be familiar with advanced OOP techniques or design patterns or anything, but you should be able to pass an introductory object oriented programming class if you want to use this system to its full potential. If you don’t have this knowledge, that’s ok. You may continue, but I will not stop to explain basic OOP concepts so you may get lost on some parts. The concepts I use are not terribly complicated so you may just pick them up by osmosis.

**A More General Engine**

This engine was not built specifically for any particular game. It’s for a longer term project that will grow more complex as time grows on and the engine will need to adapt to more features. Not only that, but my goal for a career is not to be a game designer, it’s to be a software engineer for game engines. This engine was designed in a more generic and flexible way that (hopefully) does not constrain developers to a rigid game format or feature set. Unfortunately, this can often make systems much more complicated and abstract. With great power comes great responsibility, as they say. While this engine can be manipulated in a huge, and sometimes surprising amount of ways (as I have discovered), it requires a moderately in-depth knowledge of the systems workings, which is sort of a failing on my part. Like I said however, this system is only a few months old and as it evolves, I will develop better layers of abstraction that make this unnecessary.

**I Hate Bottom Up Learning.... but it works**

The bottom-up teaching strategy is exactly the opposite of top down. In top down, you start with the basic interface of the system, teach them how to perform specific, simple actions, then begin to explain what those actions really are and what they do, exposing the low level systems. Top-down learning is good for teaching someone how to use a simple system fast. Well... this is not a simple system, and it’s not a fast project.

This is a system that you can get pretty creative with, from my experience, but it requires a working knowledge of the underlying systems and what they do (although not necessarily *how* they do it). We don’t need scripters to write mundane tasks, we need them to innovate, and having a good knowledge of the underlying systems will help immensely.

The problem with bottom-up learning is that it’s slow, and usually pretty uninteresting. You’re learning about concepts and systems that you don’t care about (yet). You get to know what they do, but not why they do it or why you should even care. Keeping this in mind, I want to treat the learning process in phases. Start with the big picture in bite-sized pieces and get into more detail later. In this way, you get a rough idea of the role that different components play and you have a preview of their purpose, which hopefully makes them a tad more interesting.

Now, here’s the good news. Bottom-up learning, from my observations, inspires creativity in most people. They learn about abstract systems in detail, and their minds can begin thinking about how they would, and what they *could* do. By the time they get to the top levels, they can use logic to reason about how the usage and class building works rather than raw memorization, which will hinder creativity and lead to bugs because of a lack of understanding of the system.

Also, as I said, this system is far from perfect, very young, and probably will of bugs. It’s often said that “top down learning is great, until something goes wrong”. When something goes wrong, the ideal interfaces that make top-down learning possible begin to collapse, and the scripter is lost on how to continue. If they understand the system in the lowest levels, they may be able to reason about what the bug is, and either circumvent or fix it.

With that out of the way, lets get goin’.

**I Intro**

**System Requirements:**

* Windows Vista or above
* DirectX 10 capable graphics card
* capable of running FMOD

**Required Software:**

* Lua 5.1
* Visual Studio 2010 Redistribution package

**Using Lua**

Do me a big favor; read this:

<http://www.lua.org/pil/index.html>

(Read sections 1 through 5, otherwise there’s nothing I can do to help you)

*The manual I linked to is for Lua 5.0, but the changes are not drastic and it’s far less dry to read.*

<http://www.lua.org/pil/#online>

Big giant list of what all lua functions do, you won’t use most of them. Refer here when you are confused about a specific piece of code.

**Object Oriented Programming in Lua**

Read these too. It’s really important:

<http://lua-users.org/wiki/ObjectOrientationTutorial>

**Brief Note about tables as classes**

You know what tables are by now because, you know, you obviously didn’t skip the text and neglect reading the manual. So you know by now that you can access elements of the table as you would data members of a class instance. You can also attach functions to tables, and call them like you would member functions of a class. With this combination, you can use tables in a way that is not a far cry from using classes in C++ or Java. When doing this, all the usual object oriented practices are good to follow. Separation of responsibilities, data and algorithm abstraction and clean, clear interfaces are essential in writing any large scale programs, and this is no exception.

When accessing data of classes, prefer to use getters and setters rather than accessing data members directly. Not only will this prevent subtle typo errors from slipping through the cracks, but when code needs to be changed, it’s far easier to change one function than it is to hop around to 20 places making sure their changes are all consistent. (UPDATE: geobox now uses a checks system that comes standard on function templates. This enables type safe code that can dramatically lower the risk of errors). Lua allows any piece of code to access any data in the global scope and that includes table data indirectly. Just because you can do it does not mean you should. In fact, I predict (and have seen many times that) you will be very sorry if you do.

**Staying out of trouble in Lua**

You probably noticed from *reading that manual*, and if you didn’t, **GO READ THE MANUAL**, that lua runs drastically different than your typical compiled language like C++. One of the biggest differences is the lack of explicit variable typing and lack of explicit variable declarations in general. Typos are your potentially your worst enemy in lua.

With actors and tables, most of the bit problems can be easily avoided simply by using getters and setters. So instead of writing this:

actor.positionX = 50

actor.positionY = 30

If you write getters and setters for the position variables, you can just do this:

actor:SetPosition(50, 30)

This will immediately solve most instances of this problem. Why? Because if you assign a value to a misspelled variable in lua, it will create that new variable with the wrong name and tell you everything is fine. If you try to call a misspelled function, it will throw a runtime error, and thus automatically catches the error rather than turning it into a subtle logic error, which could take days or weeks to show up, and a very long time to find.

Also, something small but a bit odd to get used to at first is the “syntactic sugar” for object oriented function calls with tables. When you call a function from a table, you say:

myTable:MyFunction(arg1, arg2)

Notice anything strange? It’s easy to miss. It’s the colon. You don’t use a dot when you’re calling functions from a table. You \*can\*, but the function will most likely not work correctly. The colon is actually a shorthand for passing in the table itself as a first, invisible argument into the function. Inside the function, that invisible first parameter, the table that called it, is referred to as “self”. This is effectively the same thing as the ‘this’ keyword in Java or C++. It’s an easy way to access the data contained in the calling object, or even the calling object itself from inside the function. If you use the dot instead of the colon for the function call, the “self” variable inside the called function will be nil (that’s null for most languages). You will do this sometimes, it’s an unavoidable mistake to make. When you do it, you’ll most likely get an error with a stack-trace leading to the function and the error code will say something along the lines of “attempt to index a nil object [?]”. I don’t remember the exact error at the moment, but it will be fairly obvious and will point you to the offending line, which will be the first line of code that attempts to access data of “self” or use it in arithmetic. Note that if you call a function from self, the same rules apply.

**Abstraction**

It’s important. Really, really important. The entire concept of high level programming languages are based off of re-usability and abstraction. Abstraction has a very simple meaning: the removal of detail from places it doesn’t need to be. In order to write large scale, complex programs we have to purge our own minds of details about how the underlying systems work, and only focus on the cause and effect logic of their interfaces.

**Scripting Strategies and good practices**

Going down the list here real quick:

* **Use local scope, make SURE you are in local scope**
  + This is less “good practice” and more of **“completely-necessary-practice”**
  + **use the local keyword** for **all newly declared variables** in all functions
  + keep variables in as tight a scope as possible
  + If you have a loop and need a variable to compute some data inside of it that needs to be reset every loop iteration, declare a local variable inside that loop, do NOT declare a variable outside of the loop and then repeatedly use it
  + This will actually hurt performance too if you don’t follow this rule, interestingly enough
* Use getters, setters and functions for \*everything\*
  + I cannot repeat this enough. Use functions called from tables to save yourself so much trouble from typos, changes in variable names or modifications in algorithms that you would otherwise have to hunt through the system for places that need changing
  + Not encapsulating these calls causes enough trouble in static languages as it is, but in lua, not encapsulating data can lead to subtle bugs that may go unnoticed for months, causing damage to save files and game logic
  + Even when you’re inside a method called from that table, favor getters and setters. It’s just good practice and will save you a lot of heartache, and definitely headaches.
* The only good code, is no code
  + Code is evil, don’t write it
  + Less code means less debugging, less maintenance and less complication
  + Reusability and re-factoring make things faster in the long run and a lot less frustrating
  + Make code as small as possible within reason, and as generic as possible without breaking the single responsibility principle
  + If you want to get “clever”, this is the place to do it
* Don’t use optional arguments if you can help it
  + They tend to cause a lot of bugs and confusion
  + Helpful sometimes, but mostly not worth it
  + Makes it confusing to look at function calls and tell if their arguments are correct
* Working in a typeless language
  + Do type checks on variables on function parameters if the function is not clear what type of parameters it needs
  + Mostly the typeless variables don’t cause as many problems as you would think, but don’t abuse the feature
  + Favor keeping variables to one type and do not reuse
* Follow the Single Responsibility Principle
  + Please read this about the SRP (it’s short): <http://en.wikipedia.org/wiki/Single_responsibility_principle>
  + Each function and class should do one thing, and do it well, as Mr Martin says
  + This rule is harder to follow with actors in this system, but is much easier for functions
  + Each function should have a very small, simple and well understood purpose
  + A function should be as small as it can be, and contain no more code than absolutely necessary
  + Same thing also applies for variables. DO NOT use the same variable twice for different purposes
* Follow clear and \*complete\* naming conventions
  + This is extremely important when other people, or even yourself use classes
  + Follows along the lines of the SRP
  + A function should do everything the name says, no more and no less
  + Do not abbreviate
    - Abbreviations save time typing, but cause nightmares when others or yourself try to re-interpret the code or debug
    - There is no technical reason to abbreviate, you are not limited in name length
    - for every second you save by cutting corners on typing, you cause about 5 seconds of debugging and reading time for another coder or yourself
    - Some abbreviations are fine, like standard math and programming lingo, but nothing beyond that
  + A variable should say clearly what it’s used for, and sometimes even state what range it should stay in (ie, always be 0 to 1). This helps with debugging.
* Write Pre and Post conditions on semi-complex functions
  + A pre and postcondition state not what the function does, but what the end result is and what conditions need to be satisfied for the function to work

Example:

--pre: typename is a string

--post: flags an error if the table name has already been used

function FlagErrorIfNameExistsInGlobalSpace (typeName)

if \_G[typeName] ~= nil then

PrintLn("ERROR: name "..typeName.." already exists in global space!")

end

end

* + These conditions, stated clearly and completely, help you understand the complete purpose of the function, and what needs to be true beforehand for the function to operate properly
  + If correctly used, other programmers will have a clear set of guidelines to follow for how to use your code, and what results they can expect out of it
  + Notice that they say nothing about how the function works, those should go as comments inside the function
  + preconditions can include things like certain states that a class must be in (like, must be in a certain area of the world) or a certain range that a numeric parameter must be in (like 0 to 1)
  + Post conditions should cover any effects that were inflicted or made to the state of the program. This includes what data was returned from the function and what states have been changed about the class
  + Usually these are not necessary to write for simple getters and setters
* Minimum Guaranteed Functionality:
  + Especially important in a typeless language
  + When you have a code or function that makes use of a table (class instance), that code inherently assumes that whatever object its dealing with follows a certain set of rules, has a certain set of data, and a certain set of functions. If it does not, the code breaks
  + Therefore, it is very important to ensure that, if a piece of code is going to use a series of different types of classes (which is extremely common), all of those classes share at least that certain set of functions the code assumes
    - Not necessarily the same data if you use getters and setters for everything, which you should
  + It’s good practice to include these minimum functions in the preconditions of a function
  + So, a lift of functions that the class instance must have to be used properly, and if appropriate, what those functions should do
  + SetPosition, GetPosition, ApplyForce, ect, are all obvious examples of this
* Do Not “Optimize”
  + Don’t worry about speed or efficiency
  + No seriously
  + Dont
  + Stop it
  + Right now
  + Lua is the fastest scripting language in the world, and most of what you are running will be in constant time
  + Worry about optimization only when it has proven to be a problem
  + Until then, keep it SIMPLE
  + It’s easy to take a working system and make it faster
  + It’s near impossible to take a high performance system that cuts a bunch of corners and maintain it in a safe way
  + You’re still worrying about performance
  + I’m serious, don't optimize sh\*\*
  + T\_T
  + **UPDATE:** there is **one** exception to this rule. Try to avoid excessive table creation in functions that are expected to run every frame, or more than once every frame
    - table creation uses excessive calls to malloc, which can cause performance stutters
    - prefer functions like Vec2.reuse instead of Vec2.new, if practical
    - prefer a few large tables to many small tables
* Keep things modular and independent
  + Classes and actor types should not depend on each-other
  + The only exception is when one class maintains a collection of instances of another class
  + The relationship should be clear and one-directional
  + If you must have instances of classes interact, do so through a function and make that function very tightly controlled and commented
  + Actor interactions will be a very common and difficult to fix source of bugs
  + Classes that explicitly interact in non-standard ways (ie, through the physics engine) should be written by the same person or closely collaborated with someone else to make sure that both classes are well understood in unison

**Mindsets**

* You should not think about this script system as an “engine”, whatever that means
  + It’s a library and framework to help encapsulate and simplify gameplay logic
  + Mainly to make repetitive tasks such as class creation and saving easier
  + Also, spatial, time and motion reasoning tools are included
* This engine is not “magical”, as no engines are
  + It provides a set of tools for crafting game logic, nothing more
* The world editor is for placing objects in the world, the script is for making them do stuff
  + try to keep this mentality. In general, you should avoid spawning things in script that are supposed to be permanent additions to the world, for many reasons I will cover later
* The C++ engine it runs on top of is not meant to handle gameplay logic
  + It is a computational platform that handles complex algorithms and standard, yet highly intensive tasks
  + The engine is an aid to the script, not the other way around
  + The script creates the primary purpose of the application, the engine (primarily subatoms) simply provides tools to the script

**General World Building Guidelines**

Here, I want to cover some basic guidelines that should be followed to best make use of the engine. These are things that I will cover in depth later, but for now I would like to touch briefly on them to give a preview of what’s to come.

Coarse Objects/Object Counts

The subatomic engine may look retro in its current state, but it does not use retro techniques at all. This engine was designed to handle a reasonably small number of highly complex, customizable actor as opposed to a large number of simple actor. I will cover \*how\* to do this later, but \*what\* you should strive to do is to emulate batches of small objects with larger, single objects. The lower the actor count, the faster the game will run, the easier the world will be to script, maintain and build and generally the better it will play.

Actors

In the subatomic engine, everything is classified as an actor. An actor is anything that has a conceptual presence in the world and that the game recognizes as an independent entity. Actors can contain tons of both standard and custom information to accomplish their goals and they are good for defining a wide variety of game types (enemies, the player, plants, random debris, ect).

Actors are not the “controllers” of the engine per say, but they serve as containers for controllers to the engine. They encapsulate the responsibilities of each type of object, character or anything else in a simple class that is easy to understand from the point of the high level game code. Anything that can be described as an independent entity (that is, it does not serve as a component of other entities) should be considered an actor. Actors are also the only types of classes able to make use of the engine’s object-oriented subsystem components (subatoms). I’ll explain that more later.

Tiles vs Actors

In older versions of the engine, tiles were a special type of actor that used a smaller amount of data and were more limited. However, as time went on and actors became more efficient, the cost of executing different code branches for special actor types actually grew more costly than just making them generic actors.

In the newest versions of geobox (subatomic), tiles are just a specific predefined family of classes from the base actor class. Therefore, they share the same abilities as any other actor and should be treated as such. There are special built in tools for efficiently dealing with tiles, but the engine does not treat them any differently and neither should you.

“Static” vs Dynamic

There is nothing “pre-computed” in the subatomic engine. Everything can be changed at runtime with little to no issue, albeit some things may be expensive to modify if you do it in excess (changing the scale of objects with physics properties is an example of this). Usually when I say “static”, I’m referring to actors that should stay relatively unchanged, or change only in a predictable manner, during the course of gameplay. Not just for technical reasons, but because them moving on a constant basis does not make sense.

Tiles with physical properties are a good example of this. Although they \*can\* be modified, much of their code assumes that they will not move outside of being manipulated by the editor. Also, they use cheap, static physics components inside the engine, and the physics subsystem has to do a fairly expensive rehash of geometry each time a static component moves. So while everything \*can\* be changed like a regular actor, there are some types than generally shouldn’t be.

**II**

**Inner Workings**

**A Quick Bottom-Up Tour of the Architecture**

It’s difficult to talk about things like actors without the learner knowing about how at least some of the system works on a low level, in the bottom layers. Let me run through these real quick and then explore more detail later.

Subsystems

Subsystems are the lowest level of the engine that is practical to cover from a scripting perspective, just so you can get a feel for what the engine is actually doing. A Subsystem is a collection of highly modular sets of classes in C++ that run highly complex, expensive and often very cumbersome code. Subsystems run behaviors and computations that are, in general, carried out in a specific and relatively non-customizable way in the game.

The most straightforward example of this is the physics engine. The physics engine maintains a collection of static and dynamic geometric entities that interact in ways determined by that subsystem. The computations to compute the collisions, responses and movements of these entities is highly complex, but can be handled in a generic way. They take no direct influence from anywhere else in the engine, and they do not affect anywhere else in the engine. They’re off in their own little world doin’ their thing.

A slightly less straightforward example is the lighting system. It defines a set of lights and properties of those lights, as well as a collection of shapes and geometry, also with parameters. A wealth of data is produced from this system that the engine has access to and makes use of; mostly the graphics engine with respect to direct illumination, indirect lighting, volumetric rays and shadows. Still however, the system does not make any external references to anywhere in the program, and nothing in the system influences it.

The big theme here is independence. Subsystems have no implicit interaction with anything outside of themselves. Then how are they controlled by the game? Well, that’s where subatoms come in.

Subatoms

A subatom can be viewed as link (a handle, really) to an object kept within a subsystem. When a subsystem creates an object inside of itself, it gives back a reference to the part of the program that activated it. You know how I said that objects were defined with parameters? Well, this is how you set those parameters. A subatom can be viewed as a “remote control” for the object contained within a subsystem. The subatom also allows things to view the information inside the subatom that may be a result of computations that have happened in the subsystem.

Again, with the physics engine. Let’s say that we create a physics subatom in our physics subsystem. We get back a link to this subatom. With this, we can set the object’s mass, position, rotation, scale, friction, ect, ect. Now, after we run the update function for the physics subsystem. We can look into that same subatom and get its position, which may be different than what we set it to a second ago. This is because the physics system did an update and computed, given gravity and other factors, what needed to happen to that object. The thing that hosts the subatom can then use that information to set the position of itself, its graphical sprite, it’s light, or anything else that It wants to synchronize with the physics.

Objects (Atoms)

Subatoms and subsystems normally refer to specific properties such as lighting, sound properties, visual appearance, ect. It makes sense to describe physical objects as a collection of properties with specific configurations. For instance, wooden chair has a physical shape that can interact with things (physics subatom), it casts a shadow (lighting subatom), you can punch through it with a bullet (projectile subatom) and it looks like wood and may be shiny (sprite subatom).

Furthermore, an independent, relatively simple object will usually have only one instance of these properties. The wooden chair does not cast a shadow in two different ways given the same light source and it does not appear two different ways at the same time. It then makes sense to define a standard class to describe a simple, standard object than can be kept track of as a whole.

An object (and yes, I am aware this is a somewhat poorly chosen name) has one \*potential\* copy of every subatom. It contains references to all possible types of subatoms, but they do not need to be actually linked to anything. If the object lacks a lighting subatom, it will not cast a shadow for instance, but most small objects do not need to anyway (it’s excessive and expensive to compute). An object is the primary type of entity that lets subatoms work together to form a “real” object.

Objects can provide shortcuts to a lot of standard behaviors. For instance, most objects that contain a physics subatom are physics driven, and want to align its visual sprite, character occlusion, sound emitters, sound occluders, ect directly to its physics subatom, making sure they align. Objects provide some shortcut functions for this like StdSubatomSync(), which attempts to determine what subatom properties need to be automatically set, as doing that yourself can generate a LOT of excess, very ugly code. It won’t work for everything, but most of the time it works well enough and you can add extra code after the function call to customize the results.

Actors

An object can do a lot, but it is only built to do relatively simple and standard things. Do define complex objects with interesting behaviors, we need a way to collect a series of objects and make them work together. We also need a way to abstract away the cumbersome complications of collections of objects by wrapping them in a class that is easy to understand from the point of view of the high level game logic.

Actors are essentially just a collection of objects and there is little more they have in common. Actors are mostly made up of custom data with the ability to store a group of objects, which in turn, each one stores a collection of subatoms and properties. There’s a little more to it than that in the script, but we’ll worry about that later.

The Complete Actor Model

Rather than me describing this in a confusing, abstract way, lets just look at an example: a bike, let’s say. A bike is an independent object, most would consider. It has custom properties such as maximum speed, gear settings, and various other things. The complex concept of a bike however, is actually a result of binding together various simple parts such as the body, the wheels, the handle bar. Each of these small components is considered an object. Their properties, such as their shapes, materials, paint job and sound they make when they hit each other are all defined by the configuration of their subatoms.

The subsystems take this data, some magic sh\*\* happens, and you then have a representation of these objects in the world. The configuration and properties of these simple objects together form the complex object we know as a bike. When we set the speed or press on the break of the bike, we don’t have to worry about the small components, the bike as a whole takes care of that for us. This is the idea of the actors interface.

How Actors Make the World

A collection of objects and subatoms make up an actor, and in turn, a collection of actors make up the world. The world, or area rather, is the host for all actors in that current place. The concept of an area is very simple. It’s simply a place where actors can operate.

How Tools Make the Game

Actors can have relatively cleanly defined actions and behaviors, but they are general behaviors and not well adapted to making specific series of events happen in the game world in general. In order to define a specific game experience, we must map out events that are supposed to happen, and how they must happen in a specific way. For instance, the “bridge must collapse when the player steps on the first section of it”.

In theory, we could specify all specific actions in the game by coding them outright; defining where and when they happen, and what triggers them. Discord would be very pleased at how much of a clusterf\*\*k this usually turns out to be. Trying to imagine and map out space, time and chain reactions in your head is usually a fruitless endeavour, not to mention time consuming.

To enable scripters to reason with these elements, we give tools that attempt to answer the questions: who, what, where, when, why and how. Who is naturally covered by actors, when is covered by timers, where is covered by discrete volumes, why and what is covered by events/triggers and the how is covered by a very complex tool called a path. These are very simple, general tools that can be adapted to automate more tasks as the game and script engine evolve. I will go into these in more detail later.

The Area Script

In order to keep the segments of the game modular and organized, each area runs its own high level game script. This script specifies actions that should happen based on triggers, time, events and other forms of logic. The real scripting power lies in the general lua code, and with it you can do anything you want. The tools are there to help make the job faster and the understanding of the code and events simpler.

The Final Game

The last layer is the overall game, which is primarily composed of two things: a collection of area scripts, one for each area of the game world, and a global associative array, which is used for saving and accessing information between areas. Since area scripts and sections cannot directly communicate with each other (this is on purpose), they must store their information they produce in a global associative array accessible by the engine. The other scripts can read this information and use it to determine their own actions. This can be something as simple as “the lever in section A is now in the down position”, and in area B, “if the lever in section A is in the down position, then open the door in this area.

**Subsystems**

Starting down at the bottom again, we’ll explore more in depth what each subsystem does. The idea is to learn what exactly you’re controlling before we hand you the remote control. I Think this will give you a better idea of what to expect from the systems and how to use them.

**Physics Subsystem**

It may seem like an odd choice to start with the physics system, since physics are so incredibly complicated. However, while the computations are massive, the concept of the physics system itself is probably the simplest of the whole bunch.

Purpose

The physics subsystem does what the name says: it computes lighting. Programmers always use confusing name conventions, so you should too. Nah, it computes physics obviously. The physics subsystem is for anything that needs to have reasonably correct physics interactions with the world and does not need to remain under super-tight or super-precise controls.

Physics engines look great and can provide some awesome gameplay and cinematic effects, but they are notoriously unreliable when you need strict control over what the game does. Therefore, the physics engine should be used primarily for non-essential objects and in places where a failed or unexpected action by the physics engine will not result in a no-win scenario for the player. Assume the physics engine wants to screw over the player, because it normally does.

Input

The physics systems input is a series of rigid bodies, which describes anything that has a geometric shape and does not deform. These bodies have shapes attached to them that describe their physical form and material properties. The shapes are “glued” to the bodies and operate as one rigid structure. Its pretty straight forward. Blocks, triangles, arbitrary polygons, whatever. These geometric shapes on bodies have simple properties like mass, surface friction, elasticity, rotational inertia, ect. They can either be static or dynamic.

Static bodies are things that generally stay still in the world, as the name would suggest. Although they can be changed and moved, doing so may cause strange results and some very high performance penalties. The advantage is that static bodies are very, very fast when they stay still and usually describe the majority of the world terrain. Static bodies do not have properties like mass and rotational inertia. However, they do have surface properties like friction and elasticity defined on their shapes. These parameters are fairly self explanatory.

Dynamic bodies are everything that is expecting to move frequently or unexpectedly in the world. Dynamic bodies have mass, rotational inertia, monitored forces, surface properties (like friction and elasticity). These describe the majority of moving or non-constrained actors in the game.

Physics shapes also have the concept of groups and layers. The group of a physical object is described with a single integer. If two shapes are in the same group, and neither ones group is equal to 0, then they will never collide with each other. This is useful for when you want to have multiple shapes or bodies in a single actor and you don't want them to collide.

Layers also filter collisions. A shape can be in any or all of 32 layers. What this really is, is a set of 32 bits (each one being a layer) that the shape contains, you can activate or deactivate each bit. If two shapes share any enabled bits , then they will collide, otherwise, they won’t. So, a shape with only layers 5 and 7 enabled will collide with a shape with layers 23 and 7 enabled, but will not collide with a shape with layers 2, 4, 8, 19 and 30 enabled. Shapes with all layers enabled will collide with anything except for shapes that have no layers enabled. You get the point I think. To you people who know bitwise manipulations, it’s basically:

if (shape1->layers & shape2->layers != 0)

//collision is enabled

Data Produced

The most commonly used data that is produced by a physics engine is new transformations for objects. An actor, when linked to a physics subatom, can set that subatoms position initially. This will position the subatom in the physics world, just so it’s where we want it to be at. After this, every frame the physics subsystem will update the physics world, automatically changing the subatom position, rotation and velocity based on gravity, collisions and other forces.

From then on, what normally happens is that the actor which is linked to the subatom takes the position and rotation after each update and aligns itself to it. It will also typically use this data to set the transformation of other subatoms as well such as the sprites, event subatoms, lighting shapes and other components. We normally refer to these types of actors as “physics driven”.

There is also collision information produced each time one happens. When two shapes collide, the properties of the collision (like force, impact velocity, direction, collision point) are sent to the two subatoms that own them. The actor can then in turn, on that frame, access the collision data that occurred and do whatever it wants. An impact grenade could play an explosion sound and then trigger an explosion sequence. The collision data is kept in a queue and is deleted at the start of each physics world update. So, if you want collision data, it needs to be accessed from the subatom on the same frame that it happened.

Cost of Usage

Relatively high for dynamic shapes, but relatively low for static ones. Computational complexity and power usage grows rapidly as many objects begin colliding. If most objects stay reasonably far apart and still, then the computation stays fairly cheap.

Do’s and Don'ts

* DO:
  + Use physics for cinematic sequences, interactions of background objects and anything non-essential to gameplay, but can enhance gameplay
  + Use collision data to play impact sounds, cause visual effects (like dust and sparks)
  + Apply forces to control physics objects
* DO NOT:
  + Use physics objects to script or craft gameplay sequences that affect important game logic.
    - If you have to knock over a log to make a bridge across a gap, script and animate its fall. Using the physics system will most likely result in disaster
  + Tie the player or enemies to the physics engine. You will die.
    - There is a different physics system for this, which you will use
  + Have very small objects traveling very fast
    - they will shoot through things like bullets
  + Set the position of physics subatoms every frame, or control their velocity every frame
    - Use Impulses or forces. Modifying position and velocity data directly has.... strange results

**Lighting Subsystem**

The lighting system is internally very complicated, but reasonably simple on the outside. It still has a lot of parameters and some slightly confusing concepts tied to it.

Purpose

What does it sound like? The lighting system is for calculating visual and logical lighting. Lighting can be viewed as a combination of light projection and shadows. The lighting subsystem is not only capable of producing visual light as well as shadows that are drawn to give the appearance of light occlusion, but it can also approximate the amount of light that is projected onto an object's surface, taking distance and occlusion (shadows) into account.

Input

Unlike the physics system that has a uniform type of entity, the light subsystem has 2 key types: the light and the lightObject.

Light objects are actually the simpler of the two. Since all subsystems are independent of each other, the light system does not rely on other forms of geometric data to set up the world that lights exist in. Rather, this must be explicitly defined by giving actors you want lightObject subatoms. The lightObject subatoms have a shape, an opacity and a transformation (position, rotation, scale, offset). Shape is used to determine the shape of the shadow that the light can project from it. Opacity is for determining how much light it actually blocks with its shadow [WARNING: the opacity feature when not set to 1 or 0 is not fully fleshed out and may contain bugs or unexpected results].

Lights are what they sound like. They cast light and shadow computed from the world that the lightObjects have created around them. The visual, graphic properties of lights are color, range (radius), depth range (specifies what sprites in what depth layers are lit), shadow opacity, volumetric light range, volumetric intensity and some others.

Data Produced

The light subsystem produces a wealth of visual data that is displayed in graphics, but it also produces the accumulated illumination levels for each lightObject. That is, it tells the lightObject how much light it is in total. This data can be gotten by the lightObjects host actor and used for gameplay effects like stealth, or even used to create global illumination effects. This is called logical lighting or logical casting. Lights with logical casting turned off will not be considered when computing illumination on lightObjects.

Cost of Usage

The cost of *shadows* and *volumetric light rays* are too damn high. Lights themselves are not bad. This system uses deferred shading, which is a graphics technique that makes lighting computation time independent of scene complexity. You can go ape sh\*\* with light counts and the system reacts something like honey badger. As long as you keep lights relatively small and spaced out, you can use just about as many of them as you want with little consequence.

However, shadows on the other hand are very costly to compute. It’s a geometrically intensive algorithm a generates complex geometry that must be drawn. I am going to make attempts to optimize this system further in the future. Keep lights that project shadows reasonably far apart and limit them to generally one to two that can be seen at any given time. Also, keep the number of shadow casting objects as low as possible and keep their geometry simple. They take up a good chunk of cpu power if this rule is not followed.

Volumetric light rays are very expensive because they put a high load on the GPU, but they are not dependent on scene complexity. Therefore, you should probably limit them to, one per screen, maybe two if the radius is small enough.

**Sprite Subsystem**

Purpose

Allows actors to have visual representations on screen, either animated or static.

Input

The sprite subsystem is made up of sprite subatoms and sprite resources, namely sprite sheets. Really not a lot to say here. The sprite subsystem itself provides a place to store sprites, along with resources, and provides access methods that make it easy to draw sprites in their proper depth order. Each sprite subatom, at any moment, can be assigned a “clip” to use. A clip is an animation defined by a series of frames in the sprite sheet, but also contains pivot information and geometric data so that collision detection and shadows can be synchronized with the visual sprite. [WARING: the pivot system works, but is not fully incorporated into the script yet]

Sprite Sheets

Sprite sheets are actually a tad complicated. They are not just an image with segmented frames, they are an image with a collection of frames, along with definitions of what those frames are and special position marks on each one known as pivots. They also contain geometry data about the shape of the sprite’s outline that can be used inside the engine for various purposes like projectile detection and shadow generation (as well as just general ray-tracing techniques). This data is not implicitly introduced into the other subsystems, but can be made available to them by passing them through the appropriate subatoms. Sprite sheets also contain clips, which define animations using those frames available.

The only thing you, as the scripter, really need to know is that a pivot defines the center of rotation and scaling for a sprite in local space, and the system as a whole provides an easy method for creating animations that do not need to conform to any fixed frame size. I’ll go more into detail about how sprite sheets actually work later, because they are more a thing for the artist to worry about.

Data Produced

The sprite subsystem produces visual drawings of entities with various effects and thats about it. Not much to explain in an abstract sense.

Cost of Usage

Cheap.

Do’s and Don'ts

* DO NOT:
  + assign negative scales to sprites. Use the horizontal flip property if you need to achieve this effect

**Projectile Subsystem**

Purpose

EEYUP. It’s a subsystem dedicated entirely to bullets. Well, “near instantaneous” projectiles in general. This system solves the problem of having standard, very small objects that move far too fast to reasonably and reliably detect intersections with the world given a changing framerate. Bullets can not only impact with objects, but punch through them as well (slowing down when it happens) and return a wealth of data about the intersection or impact.

Input

This subsystem is composed of a collection of projectile objects. These projectile objects do little more than collect information about when they are hit by bullets, similar to the physics subatoms with collision. They are defined by a geometric outline that is not necessarily convex. Most often, this geometry is defined by the outline of the sprite the actor is currently using. I will specify how to do this later. They work in almost identical ways. Projectile objects can spawn projectiles and can detect intersection with them. Please read the “careful” notice below, however. You can define bullet speed upon spawning them.

Data Produced

The only data produced is a list of intersections/impacts that are held in each subatom, specifying each impact that happened with each bullet. One interesting thing it computes is a distinction between an entrance and an exit

Cost of Usage

Pretty cheap. Easy to abuse however. Bullets pass so fast that there are rarely more than a few of them on screen at once, and their intersections are relatively cheap to compute. Just... don’t use it to simulate something like rain.

Be Careful

By default, when a projectile is spawned from a projectile object, it **can** hit itself and result in a very strange bug, that can be potentially hard to find if you’re not aware of this. There is however, a simple solution to this. Usually, you’ll want to **set the bullets group** to the **same group as the projectile subatom.** I would recommend using the uniqueID of the parent actor as the group number

Do’s and Don'ts

* DO NOT
  + Use non-closed geometry (ie a line)
    - For any possible line on the geometric shape, it **must intersect the shape an even number of times**
    - if this requirement is not fulfilled, the entrance/exit parameter may be incorrect.

**Event Subsystem** [**DEPRECIATED: does not operate how described anymore**]

Purpose

This is a strange one, and a bit hard to describe without specific examples. You may have heard of event driven programming. In this model, you have listeners and you have events. When events happen, listeners pick them up and respond according to what the event is and what data it contains. Turns out this is a model that solves a variety of extremely complex problems in game programming with relative ease.

The purpose of the event model is simple. The subatoms are listeners and broadcasters; that is, they are capable of sending out messages as well as receiving them. When an event subatom broadcasts an event, every other event subatom within a specified range receives this event, and stores it for the duration of that frame. The actor hosting the subatom can then receive that event data and do whatever it wants with it. The range can be infinite, so that all other event subatoms in the current world receive this message.

Why is this useful? Because actors can receive a large amount of data about what’s happening in the world without having to know who or where it came from. It’s great for defining lots of relatively simple actions and having a very easy and clean way to have actors interact. We used this system, for instance, to create interactive grass and dust that responded to fast movement from the player. In total that took about 15 lines of code to do with no explicit object interactions or messing with types.

Input

Describing the input to this system is very abstract, since there are no specific constraits on what events can be exactly. Event subatoms have a position and a radius. They also have the abillity to broadcasts messages throughout the world that the event subsystem runs. These messages (called events) contain custom data. They implicitly contain the name of the function that should be called by actors to handle the event, which is just “On” tagged on to the front of whatever their name is. So for instance, the event type “LightFlash” will have the handler function “OnLightFlash”.

There is no generic OnEvent() method. The reason for this is actually efficiency. Most actors are not interested in most types of events usually, and a generic OnEvent method would cause all actors to receive all events, which would be nearly impossible to process when the number of actors emitting events grew very large (which happens quite often).

Data Produced

The this system does not “produce” data per-say. It simply propagates custom data that was created and copies it to other appropriate subatoms in range. After each update, the listeners will have access to the messages that they have received for the duration of that frame. After the frame is passed, the event data is re-set.

Accessing this data is simple and intuitive, but when you look you look under the hood it can get a tad confusing, because it seemingly breaks modular design (it might actually lolidunno).

Quick Example

The most straightforward example I can give is an explosion. We can define an explosion event, give it a position, radius and a power. If an explosion event is broadcasted from an event subatom, then any actors with event subatoms in range will receive this event. If they happen to have a function defined on them called “OnExplosion(explosion, object)” that accepts the event data and the host object that it came from, then that function will be called automatically to handle the event. Ifffff... you didn’t catch all that, don’t worry. I’ll be explaining more in the tutorial section.

Cost of Usage

Cheap, so long as ranges for events stay fairly low, and they normally should.

Do’s and Don'ts

* DO NOT
  + Store references to events themselves
    - you can \*copy\* the data, but don’t save a link
    - Event data can be erased at unpredictable times (this is on purpose)

**Sound Subsystem**

Purpose

What does it sound like? D’ohohohohoh

Input

Similar to lights, the sound system is made up of two different subatom types: sound emitters and sound occluders. The sound emitters emit sound and the sound occlu.... you get it. Sound emitters can access sounds from files and play them. They accept parameters like volume, position and soon to come, special effects parameters.

Sound Occluders define geometric shapes that makes sound quieter when they stand between a sound source and the player. Right now, only boxes are really supported.

Data Produced

The sh\*\* that comes out of your speakers. Potentially some logical data in the future.

Cost of Usage

Short answer is I dunno. The system’s still too young and I haven’t gotten to stress test it. The long answer is, I dunno, but we’re using software processing for purposes of flexibility and comparability. The FMOD guys seemed pretty confident that their system had very good performance and speed wasn’t a concern. I haven't noticed any frame lagging yet when I overload the system with sounds by accident, but we’ll have to see with some real profiling.

**The Idea Behind Subatoms**

It seems curious that I wouldn’t just unify all of these properties under a unified structure and allow you to say, turn on and off things like shadows, light receiving, light casting ect, ect. Well, that’s what the first version of this engine did, and it sucked. There were two very major problems with it.

1. Everything was clusterf\*\*ked into a single, gargantuan class
   1. Changes to one system would often cause code to break in other systems
   2. The object class that contained all the properties became so large that not only was it very hard to manage for humans, but it also cause intellisense in VS 2010 Professional to break (!!)
   3. These properties were semi reliant on each other, making many implicit interactions that cause subtle bugs constantly
2. It was hard to customize objects
   1. Everything was auto-aligned (and there was no alternative)
      1. There were times I did not want to do this
      2. I could apply an offset, but that just made the class more cumbersome than it already was

I don’t know if subatoms are the best solution in the world to address these problems, but they allow a great deal more freedom than a unified model and thus far have been very resilient to changes in the system that made my previous version of this engine snap in half. They also are object oriented and follow the concept of aggregation/composition, which is a pretty good design from my experience. The separation of components and subsystems allowed a high level of concurrency to take advantage of multicore systems as well.

**Quick Preview of The Complete Actor Model**

Just so you’re not confused as f\*\*k while reading code snippets, let me just give you a preview of what a complete class looks like. Don’t worry about understanding this immediately, but do examine it some. It will start to make sense as we look at each component separately, but I wanted to give a quick top down view so you can see how they are constructed. It’s difficult to imagine subatoms and objects in context otherwise. I omitted some functions here for clarity. This is an example of a very simple lantern class. It is a physics driven object with 2 lights. One is a high range, low intensity light with shadows and volumes enabled. The second is a very small, high intensity light that models a light-bulb or a flame.

GeneralActor "LanternC"

{

lightR = 1,

lightG = 1,

lightB = 1

}

function LanternC:SetPosition(x, y)

self.ob.collision:SetPosition(x, y)

end

function LanternC:SetScale(x, y)

self.ob.collision:SetComponentScale(0, x, y)

end

function LanternC:SetRotation(r)

self.ob.collision:SetRotation(r)

end

function LanternC:GetPosition()

return self.ob.collision:GetPosition()

end

function LanternC:GetScale()

return 1, 1

end

function LanternC:GetRotation()

return self.ob.collision:GetRotation()

end

function LanternC:ResetForce()

self.ob.collision:ResetForce()

end

function LanternC:SyncProperties()

--set the light color

self.ob.light:SetColor(self.lightR , self.lightG, self.lightB)

end

--NOTE: we are setting the subatom positions initially to 0,0, and we change their position immediately after creation outside of the actor class

function LanternC:Initialize()

--this actor needs 2 objects, a main body and a separate “light bulb” to support an additional light source

self.ob = self:CreateObject()

self.lightBulb = self:CreateObject()

--create the sound emitter subatom

self.ob:CreateSoundEmitterSubatom()

self.hitSound = self.ob.soundEmit:AddSound(find "space.WAV");

self.ob.soundEmit:PlaySound(self.hitSound);

--create the light subatom

self.ob:CreateLightSubatom (0, 0, 38)

self.ob.light:SetRange (30)

self.ob.light:SetDepthRange (-10, 8)

self.ob.light:SetVolumetricOpacity (0.3)

self.ob.light:SetVolumetricRange (30)

self.ob.light:SetLogicalCasting (true)

self.ob.light:SetShadowOpacity (1.0)

--create the light subatom for an object to model the small light source

self.lightBulb:CreateLightSubatom (0, 0, 38)

self.lightBulb.light:SetColor (1\*10, .8\*10, .7\*10)

self.lightBulb.light:SetRange (1.0)

self.lightBulb.light:SetDepthRange (-10, 8)

self.lightBulb.light:SetLogicalCasting (false)

self.lightBulb.light:SetShadowOpacity (0.0)

--create the sprite for the actor

self.ob:CreateSpriteSubatom()

--get a clip handle for later use

self.lanternHandle1 = self.ob.sprite:CreateClipHandle(find "lantern.png")

self.ob.sprite:SetClip (self.lanternHandle1)

--the rotational inertia is infinite so it will not rotate

self.ob:CreateCollisionSubatom (0, 0, 0, 10, INFINITE)

self.collisionBox1 = self.ob.collision:AddBox (1, 1, .3, .2)

self.ob.collision:Enable ()

self.ob.collision:SetComponentScale (self.collisionBox1, 1,1)

--create the event subatom

self.ob:CreateEventSubatom(0, 0, 1)

--create the projectile component

self.ob:CreateProjectileSubatom()

self.ob.projectile:SetGroup(self.uniqueId);

end

function LanternC:Update()

self.ob:StandardSubatomSync()

--here we must synchronize the position of the light separately because it is a part of a different object, and is not synchronized in the call to StandardSubatomSync() on self.ob. Since the light subatom is the only one in the lightbulb object, there’s no need to sync anything else, so this suffices.

local x, y = self.ob.collision:GetPosition()

self.lightBulb.light:SetPosition(x, y)

end

**Tour of Subatoms**

Localized Resource References

Before we can get started, there are some slight oddities to clear up about how resources such as images and sounds are loaded and referenced by actors internally. I feel I should explain my seemingly odd, somewhat poor choice for a system here.

There are many different approaches you can take to accessing and using resources from files. One way is to have all references to any resources use strings. Not only is this highly memory and speed intensive, but it’s also prone to spelling mistakes and can make for some very serious, and hard to find bugs. Not to mention that if a resource ever changes location, you have to jump around to all references inside an actor and change every instance, and that’s definitely no fun and very time consuming.

Another approach you can take is loading in resources as self-contained objects or integer handles. The good news is, this puts resources in a centralized location. The bad news is that it puts resources in a centralized location. When file names or paths change, its very easy to go in, make changes and have your system relatively immune to spelling mistakes regarding file paths. The bad part of this is that: 1) there’s no clear way to distinguish what types of entities use what resources, especially when they are shared among multiple. 2) It breaks good modular design. Now almost all actors are referencing something outside of themselves, and if anything changes (code is more likely to change than file locations usually) it could break a large portion of your actors and you may not even know it for a while (like say, if you are developing a portion of the game where those particular actors are not used). It’s also a pain to load new resources quickly for an experimental object because you have to slog through a massive resource database and find the right spot, hopping between files.

So, what I did is kind of a mix between the two that took place in local scope. Basically how it works is that, for each class instance, usually in its initialization/creation function, you create a list of resources that that actor should be using. These references are then assigned to handles in local scope. This basically means that, anywhere inside **this instance** of **this actor**, that handle can be used to refer to resources. The handle is invalid if you attempt to use it outside of that instance. Know however, that is IS valid to share handles between *objects* (components of actors), so long as they reside in the same actor. So, in other words, local objects can use the same ones, but actors should never share resource handles. Doing so may result in nil references and various diseases.

Here’s an example of how it works. If you don’t completely get this out of context, don’t worry, we’ll explore it more in detail later:

function MyClassC:Initialize()

self.ob = self:CreateObject()

self.ob:CreateSpriteSubatom()

self:LoadResourcesAndCreateHandles()

end

--pre: these are both valid file paths

--post: -the character now contains usable handles to a walk and run animation

function MyClassC:LoadResourcesAndCreateHandles()

--create the new resource handle for a walk and run animation from the same sprite sheet

--the second string arguments define the specific animation (clip) that you want from the sprite sheet file

self.walkHandle = self.sprite:CreateClipHandle(“Sprites/enemy1.sps”, “walk1”)

self.runHandle = self.sprite:CreateClipHandle(“Sprites/enemy1.sps”, “run1”)

end

--pre: none

--post: the character is now using a walk animation and is moving forward

function MyClassC:Walk()

self.sprite:SetClip(self.walkHandle)

--do various things here to make the character move

end

This particular example is for sprite sheets and animations, but sounds work in the exact same way. I realize this system isn’t perfect, but it’s the closest I could get to a good balance between modularity, efficiency and straightforward naming. Also, its main weakness is changes to file names, which are much less volatile than code is, usually. If it proves to be a problem, it is possible to find/replace names in files.

Local Handles In General

Most of you are probably used to the concept of a handle. Although they are not particularly object oriented, they are easy to use and safe when kept to a local scope. Actors use handles a fair amount to access components that are inside their subatoms. For instance, collision (physics) subatoms can be made up of many shapes (which I called components in the function). The way we access these individual components is by storing a handle to them upon creation, and calling functions like “SetScale”, “SetFriction” or “SetElasticity” with the handle that we got during that process. So, of instance, we would say

self.triangleShape1 = self.ob.collision:AddBox(1, 1, 0.8, 0.3)

where 1, 1 is specifying the width and the height. Perhaps we discover that we want to change this scale at some point later. We would then say:

self.ob.collision:SetComponentScale(self.triangleShape1, 2, 2)

which would double the size of that shape. This strategy is common for any subatoms that are defined in pieces or have internal resources. Again, actors should not share handles between each other, as they are considered a local resource.

**Subatoms**

Now that you know the subsystems, you pretty much know what subatoms do and what they’re for. The only thing left to describe really is their current capabilities, which may all be described by functions. I’ll describe the functions by listing their pre and post conditions. I won’t bother describing the effects that are obvious

NOTE: These subatoms are not entirely complete. They are missing some functionality and may contain known, small bugs. They should allow you to accomplish most of what you need to do, however.

**The Sprite Subatom**

--standard name: sprite

function SpriteSub:SetPosition(x, y)

function SpriteSub:SetRotation(x)   
--pre: x and y are positive numbers

function SpriteSub:SetScale(x, y)

--pre: none

--post: the pivot position of the pivot is shifted by x and y on each respective axis

function SpriteSub:SetPivotOffset(x, y)

--pre: flip is either true or false

--post: the direction the sprite faces is default if false, and opposite if true

function SpriteSub:SetHorizontalFlip(flip)

--pre: depth is an integer

--post: draw order (layer) of the sprite is now ‘depth’, smaller values = further away

function SpriteSub:SetDepth(depth)

--pre:

* sheetName is a valid path and name to a .sps file
* there must be a png image located in the same directory
* if the name of the .sps file is ‘x’.sps, then the name of the image must be located in the same file and be named ‘x’.png
* clip name is the name of a valid animation held in the .sps file

--post: the opacity (the opposite of transparency) has been set to the level specified

function SpriteSub:CreateClipHandle(sheetName, clipName)

--pre: opac is between 0 and dd1

--post: the opacity (the opposite of transparency) has been set to the level specified

function SpriteSub:SetOpacity(opac)

--pre: r, g, and b are all floating point values above 0

--post: tints the sprite to the colors specified by multiplication on each channel

function SpriteSub:SetTonemap(r, g, b)

--pre: fps is above 0

--post: speed of the animation playing has been set to display ‘fps’ frames per second

function SpriteSub:SetFps(fps)

--pre: clipHandle is a valid handle was was retrieved locally in the actor

--post: speed of the animation playing has been set to display ‘fps’ frames per second

function SpriteSub:SetClip(clipHandle)

--pre: SetClip was previously used to set the subatoms clip to a valid local clip handle (same precoindition for the next 3 functions)

--post: speed of the animation playing has been set to display ‘fps’ frames per second

function SpriteSub:Play()

--post: the animation is paused

function SpriteSub:Pause()

--post: the animation is paused and its elapsed time is set back to 0

function SpriteSub:Stop()

--pre: SetClip previously used to set the subatoms clip to a valid local clip handle

--post: returns number of seconds of the animation played, taking pauses into account

function SpriteSub:GetSecondsPlayed()

function SpriteSub:SetAnimationProgress(p)

--pre: SetClip previously used to set the subatoms clip to a valid local clip handle

--post: returns number of times the animation has looped

function SpriteSub:GetNumTimesClipPlayed()

**The Light Subatom**

Casts visual light, logical light and shadows, as well as volumetric light

**Notes:**

Reasonable Ranges for light intensity: 1 to 5 for each color channel

--standard name: light

function LightSub:SetPosition(x, y)

--pre: r, b and g are positive values

--post: visual light now casts only onto sprites within the depth range

function LightSub:SetColor(r, g, b)

--pre: low and high are both integers, high >= low

--post: visual light now casts only onto sprites within the depth range

function LightSub:SetDepthRange(low, high)

--pre: r is a positive number

--post: light radius has been ser, the attentuation will reduce to 0 at distance r

function LightSub:SetRange(r)

--pre: g is an integer

--post: the lights group is now set to an integer value and no light objects with that same value will be logically lit, except if g = 0

function LightSub:SetGroup(g)

--pre: o is between 0 and 1  
--post: Set the intensity of the light rays on screen

function LightSub:SetVolumetricOpacity(o)

--pre: o is a positive number

--post: Set the world space radius of the volumetric light

function LightSub:SetVolumetricRange(o)

--pre: o is between 0 and 1  
--post: Set how much light bias shadows from this source have

Setting o to 0 will turn off shadows for this light

LightSub:SetShadowOpacity(o)

--pre: b is true or false

--post: Set if the accumulation level of light objects is modified by the light

function LightSub:SetLogicalCasting(b)

**The LightObject**

Allows generation of shadows from lights and can measure light illumination projected onto it.

--standard name: lightObject

function LightObjectSub:SetPosition(x, y)

--pre: x and y are positive numbers

function LightObjectSub:SetScale(x, y)

function LightObjectSub:SetRotation(x)

--pre: b is true or false (boolean)

--post: the normals of the geometric outline are inverted, causing shadows to project from the front or the back-side of the geometry, depending on what the default was

function LightObjectSub:SetFrontFaceProjection(b)

--pre: width and height are positive numbers

--post: a box has been added to the light object and will be included in occlusions

function LightObjectSub:AddBox(x, y, width, height)

--pre: min.x > max.x, min.y > max.y

--post: a box has been added to the light object and will be included in occlusions

function LightObjectSub:AddBox(min, max)

--pre: v1, v2, v3 are Vec2 vertices mapped in a counter-clockwise order

--post: a box has been added to the light object and will be included in occlusions

function LightObjectSub:AddTriangle(v1, v2, v3)

--pre: g is an integer

--post: the lightObjects group is now set to an integer value and no light objects with that same value will be logically lit, except if g = 0

function LightObjectSub:SetGroup(g)

--pre: none

--post: returns 3 results, the r, g, b intensity of illumination projected on a light object as a summation of all lights

function LightObjectSub:GetIllumination()

--pre: o is 0-1

--post: the shadow opacity of the lightobject has been set to o

function LightObjectSub:SetOpacity(x)

--pre: the parent object has a valid sprite subatom and it is set to a valid clip

--post: the projection/shadow geometry is now set to

--[this function is not reliable yet]

~~function LightObjectSub:SetProjectionGeometryToSpriteGeometry()~~

--pre: light analytic geometry was not previously set to something else

--post: the accumulation of light on the object is now computed by the projection of light onto the surface of a circle with radius r

function LightObjectSub:SetQueryGeometryToCircle(r)

--pre: light analytic geometry was not previously set to something else

--post: the accumulation of light on the object is now computed by the projection of light onto the surface of a line in the local coordinates {localA, localB}

--[this function is not reliable yet]

function LightObjectSub:SetQueryGeometryToLine(localA, localB)

**The Collision (physics) Object**

Allows input and collection of output regarding the physics engine. Actors or objects using enabled physics subatoms should limit the amount of direct position setting they do. This may cause strange results.

--standard name: collision

function CollisonSub:SetPosition(x, y)

function CollisonSub:SetRotation(r)

function CollisonSub:SetVelocity(x, y)

--pre: width > 0, height > 0, elasticity is 0-1, friction > 0

--post: a box shape has been added to the collision shape at the center of the body.

--Returns a handle to the newly created shape

function CollisonSub:AddBox(width, height, elasticity, friction)

--pre: radius > 0, elasticity is 0-1, friction > 0

--post: a circle shape has been added to the collision shape at local coordinates (offsetX, offsetY).

--Returns a handle to the newly created shape

function CollisonSub:AddCircle(radius, elasticity, friction, offsetX, offsetY)

--pre: v1, v2, v3 are mapped on the plane in a clockwise order, elasticity is 0-1, friction > 0

--post: a triangle shape has been added to the physics body.

--Returns: a handle to the newly created shape

function CollisonSub:AddTriangle(v1, v2, v3, elasticity, friction)

--pre: physics body must contain shapes before being enabled

--post: physics body is enabled and will now be simulated

function CollisonSub:Enable()

--pre: physics body must contain shapes before being enabled

--post: [The precise consequence of this function is not well understood. Don’t use it]

~~function CollisonSub:EnableAsRouge()~~

--pre: physics body must contain shapes before being enabled

--post: physics body is enabled and will now be simulated

function CollisonSub:Disable()

--pre: shapeIndex is a valid handle to a shape in the body

--post: the surface velocity of the shape has been set

function CollisonSub:SetSurfaceVelocity(shapeIndex, x, y)

--pre: shapeIndex is a valid handle to a shape in the body, x > 0, y > 0

--post: the surface velocity of the shape has been set

function CollisonSub:SetComponentScale(shapeIndex, x, y)

--Returns: x, y velocity speeds

function CollisonSub:GetVelocity()

--Returns: body rotation between 0 and 2\*pi

function CollisonSub:GetRotation()

--Returns: body mass in kg

function CollisonSub:GetMass()

--post: Applies impulse to body at an offset from the bodies center

function CollisonSub:ApplyImpulse(velX, velY, offX, offY)

--post: Applies lasting force to body at an offset from the bodies center

function CollisonSub:ApplyForce(velX, velY, offX, offY)

--post: sets forces and velocity to 0

[I am aware this function is badly named and plan to change it]

function CollisonSub:ResetForce()

--pre: layer >= 0 and layer <= 31, is an integer

--post the layer corresponding to ‘layer’ has been enabled

function CollisonSub:EnableLayer(layer)

--pre: layer >= 0 and layer <= 31, is an integer

--post the layer corresponding to ‘layer’ has been disabled

function CollisonSub:DisableLayer(layer)

--pre: layer >= 0 and layer <= 31, is an integer

--post all layers have been disabled except for ‘layer’

function CollisonSub:SetLayer(layer)

--pre: none

--post all layers have been enabled

function CollisonSub:EnableAllLayers()

--pre: none

--post all layers have been disabled

function CollisonSub:DisableAllLayers(layer)

**Projectile Subatom**

--standard name: projectile

function ProjectileShapeSub:SetPosition(x, y)

function ProjectileShapeSub:SetRotation(x)

--pre: x > 0, y > 0

function ProjectileShapeSub:SetScale(x, y)

--pre: d > 0

--post: sets the objects resistance to penetration

function ProjectileShapeSub:SetDensity(d)

--pre: g > 0

--post: if g != 0, then any projectile that has the same group value will not intersect

function ProjectileShapeSub:SetGroup(g)

--pre: the host object has a valid sprite subatom and it is set to a valid clip

--post: the geometry used to detect intersections has been set to the geometry specified by the sprite outline

function ProjectileShapeSub:SetProjectileGeometryToSpriteGeometry()

--pre: speed > 0, group >= 0

--post: projectile has been spawned at (ix, iy) and will terminate at (tx, ty)

function ProjectileShapeSub:SpawnProjectile(initX, initY, termX, termY, speed, group)

--pre: width > 0, height > 0, the edges will not overlap with previous shapes

--post: adds a box to the shape collection

function ProjectileShapeSub:AddBox(x, y, width, height)

--pre: min.x < max.x, min.y < max.y, the edges will not overlap with previous shapes

--post: adds a box to the shape collection at local coords (min.x, min.y), max.x, max.y

function ProjectileShapeSub:AddBox(min, max)

--pre: v1, v2, v3 are Vec2 vertices mapped in a counter-clockwise order, edges will not overlap with previous shapes

--post: adds a triangle shape to the collection

function ProjectileShapeSub:AddTriangle(v1, v2, v3)

**Sound Emission Subatom**

--standard name: soundEmit

--pre: r > 0, r < 255

--post: the overall volume of all sounds playing is

function SoundEmitSub:SetVolume(r)

--pre: none

--post: sets the position of the sound source

function SoundEmitSub:SetPosition(x, y)

--pre: ‘filename’ is a valid file location of a sound resource of any supported format of FMOD

--post: the sound can now be referenced internally by the actor and can be played

--returns a local handle to the sound resource

function SoundEmitSub:AddSound(filename)

--pre: handle is a valid local handle to a sound resource

--post: the sound linked to by handle is played and effects are applied based on the emitters configuration

function SoundEmitSub:PlaySound(handle)

**Sound Occlusion Subatom**

--standard name: soundOcc

[NOT FINISHED, DO NOT USE]

function SoundOccSub:SetAsSurface(width, dis, rotation)

function SoundOccSub:SetPosition(x, y)

function SoundOccSub:SetRotation(r)

function SoundOccSub:SetScale(x, y)

function SoundOccSub:SetDirectOcclusion(r)

function SoundOccSub:SetReverbOcclusion(r)

**Objects (Atoms)**

As said, actors are composed of objects, and objects are composed of subatoms. Stepping up one level in the system, we now cover objects in more detail.

Really, since we’ve covered most of the conceptual stuff of objects, there’s not a lot left to say about them by themselves. The easiest thing to do is just to show you the definition of what they are:

Object =

{

actorIndex = 0,

objectIndex = 0,

collision = nil,

lightObject = nil,

light = nil,

sprite = nil,

projectile = nil,

char = nil,

charOcc = nil,

soundEmit = nil,

soundOcc = nil

}

function Object:GetPosition()

function Object:SetPosition(x, y)

function Object:GetScale()

function Object:SetScale(x, y)

function Object:CreateCollisionSubatom(x, y, rot, mass, rotInertia)

function Object:CreateSoundEmitterSubatom()

function Object:CreateSoundOcclusionSubatom()

function Object:CreateStaticCollisionSubatom(x, y, rot, mass, rotInertia)

function Object:CreateEventSubatom(x, y, radius)

function Object:CreateLightObjectSubatom()

function Object:CreateLightSubatom(x, y, range)

function Object:CreateSpriteSubatom()

function Object:CreateProjectileSubatom()

function Object:CreateCharacterSubatom(x, y)

function Object:CreateCharOcclusionSubatom(x, y)

function Object:StandardSubatomSync()

That’s it really. This is a heavily abstracted version of the C++ based object, but this is pretty much the entire code in lua (with function bodies excluded). The data you see contained in the class consists of host information about the actor that contains the object, and a potential reference to all possible subatom types; at most one per object. Just to be clear, in those nil references may as well not be there given how lua is compiled. They are just there to make the potential reference names clear.

**Creation and Manipulation of Subatoms**

It’s not really that complicated. From the object contained in the actor, you call one of the functions given above. From then on, a subatom will be contained in that function that uses their standard name. So if we call, from inside an actor:

self.ob:CreateCollisionSubatom(0,0,0,1, 3)

From then on, the object ‘ob’ will contain a subatom called collision, which we access like this:

self.ob.collision

We can also call functions from it like this:

self.ob.collision:SetPosition(60, 2)

Use the “standard names” from the subatom section to access the subatoms that have been created.

**Object Synchronization**

One of the most potentially confusing and cumbersome tasks when dealing with objects is synchronization of their subatoms. Basically, this means that things such as the sprite and collision shapes stay aligned, the shadows match the outline and position of the sprite shape, ect. This involves a huge amount of raw work and code. It’s definitely no fun and very prone to bugs.

While I could not find a really \*good\* way to synchronize then in a standard way while maintaining the freedom that subatoms are supposed to allow, I did manage to create a function that automates most of the repetitive tasks for most objects. This is called StandardSubatomSync(). It’s a rather complicated function, but it can really only does one thing.

By default, as you can see, objects have a concept of position and rotation. All subatoms that have these concepts will be aligned to both the properties when this function is called. Notice that scale and offset are not touched, because the automatic synchronization of these stats for anything other than very simple objects will usually produce undesired results, and many of the subatoms do not have a concept of offset.

This call is not meant to be a cure-all for synchronization and will not produce desired results if you want a subatom to be slightly different than the object's transformation. To override this you can simply call the subatom’s specific property setting function after the sync call. So, in other words, if we wanted our lanterns light to be positioned 10 units to the right of the actual lantern object, we would simply say in the update function:

function LanternC:Update()

--align the object, event, light, and sprite subatoms with the collision subatom (pos and rotation)

self.ob:StandardSubatomSync()

local x, y = self.ob:GetPosition()

self.ob.light:SetPosition( x + 10, y)

end

**Synchronization with Physics Driven Objects**

What happens if there is a collision subatom enabled on the object? In almost all cases, you will want an object that has a physics subatom to be physics driven, that is, almost completely controlled by the results of the physics interactions. So, if the object has a collision subatom, and the subatom is currently enabled, the object’s status will be aligned to the physics subatom, and THEN it will run the normal synchronization call. The end result is that the other subatoms will be aligned with the collision subatom.

**When to Create Multiple Objects**

It’s best, if you can help it, to compose actors of only one object. It makes things simpler. Sometimes however, they need to be composed of multiple. So when should actors be composed of more than one object? The answer is actually pretty much common sense: because objects can only support one instance of each subatom type, you should have more than one object in an actor when you need two or more subatoms of the same type.

Take the lantern class for instance. I needed a light source that was far-reaching and soft, and a light source that was very small and extremely bright. So I created an object for the main body and light source of the lantern, and then I created another object to serve as a “light bulb”. The light bulb object only hosted a light, because that was all it needed.

**Composing Actors From Other Actors**

I do not recommend this usually. It can be done easily, but it adds a form of “memory management” to the process of actor creation. Usually, all subatoms and objects are cleaned up when an actor is deleted. Since actor composition is not common enough to warrant muddying the architecture with concerns over it, actors must be managed, created and released manually. This is perfectly fine to do and is a good practice in object oriented programming, but you must be careful to release

**Actors**

So, you now can see how objects work to bind together subatoms to create a single entity with customizable properties that interact with the world in a complex way. An object alone, however, is not enough to define anything really interesting, especially in terms of gameplay. Also, while it has been hidden from you thus far, accessing engine subsystems with subatoms is a fairly messy process that is not intuitive to write.

What we need is another container that stores multiple objects, allows natural and easy access to engine subsystems and can be referenced in an intuitive way in the script. Ideally, it will describe anything that the game would consider as an independent object that does not describe a component to another, larger object.

There are different “levels” of actors that serve various purposes and automate tasks that are repetitive, time consuming and unnecessary to perform yourself.

**The Base Actor**

All actors are base actors; that is, they all inherit from the base actor class. Their main purpose is to allow storage of collections of objects and abstract away the details of accessing the C++ based engine. They also allow a way to localize resources by working with the databases in the engine, making access to sprite sheets, sounds and potentially other resources easy to load and access from independent classes. base actors also automatically handle resource cleanup and reference removal from the engine when actors are removed from the world.

Event Calling

One function that base actors provide is the ability to automatically handle events that are detected by their event subatoms, if they have them. The process of how they collect this data is a tad intuitive, and the way to handle these events is easy, but does not fit naturally with the rest of the design pattern, and is thus slightly out of place. Never the less, events can be very easily detected and responded to with minimal code because of the base actor. I will cover this topic more in depth later.

Automatic Updating

Although it seems trivial, having a function that is automatically called for each actor every frame is extremely useful and a natural way to deal with changing status and movement of actors. All actors can have an Update() or a StaticUpdate() function. Updates are called every frame, while static updates are called only when something has made a known change to that actor when it wasn’t usually supposed to move or have its properties modified. Update functions are where most of the “action” of the game is enabled on a per-actor basis. It’s also normally where actor-object-subatom synchronization takes place.

**The General Actor**

The general actor does many things:

1. Provides a function called GeneralActor() that makes it much easier to declare new class types.
2. Provides automatic inheritance from the BaseActor class
3. Provides the scripter with the ability to give new actor types a generic set of simple properties which can be:
   1. preserved in files automatically
   2. manipulated by and blended by animations
   3. accessed or manipulated by triggers and events
   4. used internally like normal variables
4. Allows binding to animation paths
5. Ensures that new actor types own a standard set of functions that actors must have to be used correctly by the editor and tools
6. Provide a generic save function that preserves general behavior, identity and properties
7. Provide a generic copy function that duplicates general behavior and properties (not identity, of course)

The Property Table:

[This feature is still in development, but appears to be stable. There will be more features to it as time goes on]

The properties table is a list of data that needs to be recognized as an integral part of a class type. They appear as part of the declaration of a new custom actor type. Properties are extremely convenient because they are automatically handled by the engine in many ways. For one, their instance values are automatically preserved in save files, so actors can be saved exactly as-is within as much detail as you want.

Properties can also be manipulated by animations. In animation tracks/paths, you can set the nodes on each key-frame so that they will be blended between on each section of the track. In this manner, animation tracks can modify any custom property of any actor, which is extremely convenient for defining stuff like light animations, volume/reverb changes in sound, sprite frame progression, ect.

The current downside to properties is that they must be primitive data. No references, no functions, no tables. Only numbers, strings, booleans and other primitive types. This may change in the future to include certain kinds of tables. If you want to define this type of data in an actor, do it during the initialization function, not the property table declaration. This is admittedly semi-awkward, but not terribly out of place in lua.

Class Definition and (some) Functions

Here is the definition of the class, along with a description of functions you may need to interface with and what they do. I will not Describe the animation functions here, because I have not described animations yet.

[NOTE: the animation functions are still under development and not quite fully functional]

GeneralActorC = {properties = {}, --describes primitive class properties

trackData = {} --describes relationships to animations

}

--pre: subset is a table, where all of its keys are in the property table of the calling actor

--post: the properties in the calling actor that have the names indicated by the keys of the subset table

function GeneralActorC:SetPropertiesSubset (subset)

--pre: ‘name’ is a string denoting a valid name that is not currently assigned to another actor

--post: the actor can now be referenced by that name

function GeneralActor(name)

--pre: none

--post: produces a string which will re-create the actor and its property states upon execution

--returns: serialization of class components in string form

function GeneralActorC:Save()

--pre: (x and y) or numbers or (x and y) are nil

--post: produces an instance of the specific class that shares the same behavior, transformation and property values

function GeneralActorC:Copy(x, y)

--[Animation Functions]

function GeneralActorC:AddTrack(trackRef)

function GeneralActorC:UpdateTracks()

function GeneralActorC:PlayTracks(...)

function GeneralActorC:PauseTracks(...)

function GeneralActorC:StopTracks(...)

function GeneralActorC:ClearTracks(...)

function GeneralActorC:GetTrackTime(...)

function GeneralActorC:SetTracksPositionState(state, ...)

function GeneralActorC:SetTracksRotationState(state, ...)

function GeneralActorC:SetTracksScaleState(state, ...)

function GeneralActorC:SetTracksOffsetState(state, ...)

function GeneralActorC:SetTracksLoop(state, ...)

**The Custom Actor**

Any custom actor that scripters make should be an extension of the general actor. The custom actor is not defined anywhere in the script engine. Rather, it is defined by each scrpiter to make custom classes for the purposes of the game, as the name would imply.

The custom actor must follow a set of rules that are described by the general actor above. Use the Lantern class that I posted beforehand for reference. Here it is again just so you don’t have to constantly flip back and forth.

Actually, I want to break it up and analyze each piece so you can see the full anatomy:

***(A)***

GeneralActor "LanternC"

{ <-------------------- this defines the properties table

lightR = 1,

lightG = 1,

lightB = 1

}

This is the custom actor class declaration.

There are 3 main parts. The “GeneralActor” keyword (which is actually a function) which specifies that the name which comes next should be created into a new actor type. The Table that comes after the name is the properties table. As I said before, **you can use only primitive data types**. No userdata, no functions, no tables or references.

**UPDATE:** table properties are now supported, but have not been extensively tested. Use with caution, and do not use on tables with metatables (unless \_\_serialize is a specifies metamethod) or actors that are complex (other actors, for instance). More precisely, use only with table types that can be easily serialized.

After the type declaration, we can begin to define behaviors and functions:

***(B)***

function LanternC:SetPosition(x, y)

self.ob.collision:SetPosition(x, y)

end

function LanternC:SetScale(x, y)

self.ob.collision:SetComponentScale(0, x, y)

end

function LanternC:SetRotation(r)

self.ob.collision:SetRotation(r)

end

function LanternC:GetPosition()

return self.ob.collision:GetPosition()

end

function LanternC:GetScale()

return 1, 1

end

function LanternC:GetRotation()

return self.ob.collision:GetRotation()

end

function LanternC:ResetForce()

self.ob.collision:ResetForce()

end

These are (some of the) standard transformation functions that each actor must fulfill (there’s an offset function that I have omitted). These standard getters and setters force the scripter to implement the concepts of transformation that most code needs to operate on actors in a meaningful way, the most obvious example being animations. The ResetForce() one is a bit of an oddity. This function is only meaningful for actors that use physics subatoms, but its non-the-less very important because manipulations of actors may cause strange behaviors within the physics engine.

Animations for instance, will set the position of an object every frame, usually off the ground. As gravity builds on the object, it starts to accumulate speed, but is held in place. If the animation had no way to reset the force on the actor and it was controlled by the physics subatom, the actor would jitter along the animation and shoot off like a rocket when it was released from the tracks influence.

I will attempt to make future improvements to automate this task, but for now, actors must implement a function of this nature.

***(C)***

function LanternC:SyncProperties()

--set the light color

self.ob.light:SetColor(self.lightR , self.lightG, self.lightB)

end

This is another odd, yet very important one. It was mostly made for paths and animations. Since animations cannot modify subatoms directly, and they can only modify the properties table of actors, there should be a convenient way to have the changes to the properties immediately take effect on the subatoms, objects and whatever else they may need to. This function generally serves that purpose. You can see that we’re just setting the light’s illumination value to the color values held in the property table we declared above. This function is called automatically after every frame of animation and will be called in the future by other tools as well. Feel free to invoke this function yourself at any point if you plan on using properties as your primary way of setting states in the actor (which may not be a bad idea).

***(D)***

--NOTE: we are setting the subatom positions initially to 0,0, and we change their position immediately after creation outside of the actor class

function LanternC:Initialize()

--this actor needs 2 objects, a main body and a separate “light bulb” to support an additional light source

self.ob = self:CreateObject()

self.lightBulb = self:CreateObject()

--create the sound emitter subatom

self.ob:CreateSoundEmitterSubatom()

self.hitSound = self.ob.soundEmit:AddSound("space.WAV");

self.ob.soundEmit:PlaySound(self.hitSound);

--create the light subatom

self.ob:CreateLightSubatom (0, 0, 38)

self.ob.light:SetRange (30)

self.ob.light:SetDepthRange (-10, 8)

self.ob.light:SetVolumetricOpacity (0.3)

self.ob.light:SetVolumetricRange (30)

self.ob.light:SetLogicalCasting (true)

self.ob.light:SetShadowOpacity (1.0)

--create the light subatom for an object to model the small light source

self.lightBulb:CreateLightSubatom (0, 0, 38)

self.lightBulb.light:SetColor (1\*10, .8\*10, .7\*10)

self.lightBulb.light:SetRange (1.0)

self.lightBulb.light:SetDepthRange (-10, 8)

self.lightBulb.light:SetLogicalCasting (false)

self.lightBulb.light:SetShadowOpacity (0.0)

--create the sprite for the actor

self.ob:CreateSpriteSubatom()

--get a clip handle for later use

self.lanternHandle1 = self.ob.sprite:CreateClipHandle("Scripts/Classes/Lantern/lantern.png")

self.ob.sprite:SetClip (self.lanternHandle1)

--the rotational inertia is infinite so it will not rotate

self.ob:CreateCollisionSubatom (0, 0, 0, 10, INFINITE)

self.collisionBox1 = self.ob.collision:AddBox (1, 1, .3, .2)

self.ob.collision:Enable ()

self.ob.collision:SetComponentScale (self.collisionBox1, 1,1)

--create the event subatom

self.ob:CreateEventSubatom(0, 0, 1)

--create the projectile component

self.ob:CreateProjectileSubatom()

self.ob.projectile:SetGroup(self.uniqueId);

end

This is the initialization function that is automatically called upon creation of an instance of a new actor. Everything that the class needs upon startup should go in here. Now listen, this is very important:

***\*All\* data members we need that cannot or should not be put in the properties table should be initialized and created here***

[NOTE: I am working on enabling the inclusion of more complex data types in the properties table, but references or functions will never belong there]

Although it may seem like an awkward practice to “declare member variables” in an initialization function, we’ve already been doing this all along with subatoms and objects, they’re member variables as well. In general, anything that the class will need during runtime that is not a basic property (of a primitive data type) should be declared in the initialization function.

***(E)***

function LanternC:Update()

self.ob:StandardSubatomSync()

local x, y = self.ob.collision:GetPosition()

self.lightBulb.light:SetPosition(x, y)

end

Finally we come to the all-important Update() function. As I have stated, this is where most of the action and synchronization takes place. This function is called automatically for every actor on every frame. You should keep them relatively small for clarity. Imagine the “Update” function a lot like the main() function of a program. In general, you should not put complex code logic in main because it gets clusterf\*\*ked really fast. The same holds true for these.

Now, let’s explore an alternate function here:

***(E)*** *(2)*

function LanternC:StaticUpdate()

self.ob:StandardSubatomSync()

local x, y = self.ob.collision:GetPosition()

self.lightBulb.light:SetPosition(x, y)

end

This is that “StaticUpdate()” I was talking about earlier. The only difference between a static and regular update is that static updates are not called every frame, but invoked by specific events in the code. This includes things like the world editor moving an actor, an animation operating on it, or various other things that cause the actor to change state in some way. Any actor that has a static update, rather than a dynamic update, should not, in general, change itself. In other words, our lantern now becomes something more akin to a torch on the wall. But.... then it would also need a static physics subatom rather than the dynamic one it has now... and... you know what? That was a terrible example, MOVING ON!

All actors should have either a static or dynamic update, but not both. If they have neither it is very difficult to keep them sync’d with the game, and if they have both, the static update is essentially redundant and may cause subtle bugs.

**Creating/Spawning Actors**

Now that we can create new actor types, we need a way to actually create new actor instances, or instances of those types. They will make up the main game world. Spawning actors is an easy feat, but there’s a few ways to do it and a few things to watch out for. We’re going to stick with our good ol’ lantern class as an example.

The most flexible way to declare a new, nameless actor is with the following line of code.

LanternC.new(x, y) --x and y is where you want to place the new actor in the world

***UPDATE****: initialization parameters are being phased out due to incompatibility with the editor*

This line actually lets you do quite a few things. The function itself returns a reference to the newly created object, and you can bind it in local or global scope. This line also does not require you to store a permanent, or even reference to the new actor. You just spawn it ant leave it if you want. You can also store a temporary reference just to manipulate it.

function SpawnLantern(x, y, rot, r, g, b)

local lantern = LanternC.new(x, y)

lantern:SetRotation(rot)

lantern:SetLightColor(r, g, b)

end

The reference “lantern” does not exist after the function exits, but the lantern lives on in the world. Of course, there’s no direct way to access this lantern anymore from the script, which may be perfectly fine. You’ll create it, shape it up into what you want it to be, then abandon it. It’ll be like “FINE, JUST LEAVE ME HERE WHY DON’T YOU!?”, and you’ll be like “ok”.

**Spawning Actors with Names**

Sometimes though you want to keep tabs on the bastard because he owes you a bunch of money or something, so you need a way to find him any time after spawning. In this case, the actor should have a name.

We can set the name of an actor with the BaseActor:SetName() function that is apart of all actors. When we set the name of an actor, it not only gives the actor a string to identify itself with, but it also creates a reference to that actor in global space by the same name. Now... it’s actually possible to bypass naming the actor and just giving it a global reference by doing the following:

lantern1 = LanternC.new (x, y) --assume the parameters are defined somewhere

lantern:SetRotation (rot)

lantern:SetLightColor (r, g, b)

This... however is really awkward and kind of dangerous. The reason being that this actor now \*cannot\* have a name that matches its global reference. There are two ways you can try to do this:

lantern1 = LanternC.new (x, y)

lantern1:SetName(“lantern1”) //UPDATE: no longer supported

Because names need to be unique, when an actor is assigned a name, it does a security check to see if that name already exists in the global scope... and clearly it does.... Therefore, this method will not work at all.

The other method you can do is this:

LanternC.new(x, y):SetName(“lantern1”) //UPDATE: no longer supported

This will work, it’s just awkward and cumbersome to write. Not only that but the combination of lines:

LanternC.new(x, y):SetName(“lantern1”) //UPDATE: no longer supported

lantern1:SetRotation (rot) //UPDATE: no longer supported

lantern1:SetLightColor (r, g, b) //UPDATE: no longer supported

Is not exactly an intuitive logical flow for someone new to the script system, trying to analyze it. So, I created a shortcut syntax:

\_LanternC "lantern2" //UPDATE: still supported, but I don’t recommend using it

lantern2:SetRotation (rot)

lantern2:SetLightColor(r, g, b)

You just type an underscore, then the name of the actor type, then the name of the actor instance. From then on, you can reference this actor throughout the program. I think this is easy enough, just watch out for that underscore, it will take a bit of getting used to.

**Names as Strings**

This was a feature implemented mostly for save files, but it’s proven to be a fairly useful tool in general.

--pre: ‘name’ is the string name of an existing actor in global space

--post: returns the reference to the actor

GetActorRef(name)

Now... the observant reader will be wondering why on earth I felt the need to re-create the operation of \_G[“name”].... which allows you to access global variables by string names. Well... you’ll see in a second. In general though, you should **avoid using strings to reference actors** simply because it is outrageously expensive to do.

**Indexed Names [DEPRECIATED]**

**[UPDATE: this feature turned out to be totally useless and is no longer supported]**

One profoundly useful feature that actors have is the ability to give themselves indexed names. An indexed name is what it sounds like: it’s a name with brackets and an index, just like an element of an array. In fact, that’s exactly what it is! Actors that have the same base names with different indices are grouped into arrays (really tables), although this process is hidden from the you, the scripter. You just get to enjoy the benefit of it!

Lets see what we can do with this:

\_LanternC "lantern[2]"

\_LanternC "lantern[3]"

\_LanternC "lantern[4]"

\_LanternC "lantern[5]"

for i = 2, 5 do

lantern[i]:SetPosition(-10 + i, -3)

end

This code will spawn 4 lanterns, and then we can use a for-loop below to set their positions in the world in a row, and we access them just like an array. Notice how you don’t have to create an array to store them, the SetName() function automatically detects when a name is indexed and makes sure that a table exists that it able to store them.

Now, unfortunately, since my magic little syntax there is an exploitation of how lua’s native syntax works, it was bound to catch up with me sooner or later.

for i = 2, 5 do

\_LanternC "lantern[“..i..”]" ←------ this will not work

end

for i = 2, 5 do

lantern[i]:SetPosition(-10 + i, -3)

end

If we want to declare a sequence of objects in a loop in the same fasion, the shorthand will actually not work anymore as-is, but it will with a small modification:

for i = 2, 5 do

\_LanternC("lantern[“..i..”]") ←-------- this will work

end

for i = 2, 5 do

lantern[i]:SetPosition(-10 + i, -3)

end

Not quite as convenient, but it’s a minor issue I’m looking into fixing it.

**Using Indexed Names [DEPRECIATED]**

Indexed names do not work as a generic table. You can only use integers to index the actors, not strings or anything else. Also, I do not recommend using a regular for-loop with an integer counter.

for i = 2, 5 do

lantern[i]:SetPosition(-10 + i, -3)

end

I used to do this, but then I took an error to the knee. You can get away with this sometimes, but its not really a good idea. If an actor gets deleted, or a reference gets set to nil, you’re kinda boned. Your code will suddenly access a nil or dead actor and throw an exception. Even if this doesn’t happen or you guard against it, it’s difficult to keep track of the upper bound of the array as it may change unpredictably. What you should do is this:

for k, v in pairs(lantern) do

lantern[k]:SetPosition(-10 + i, -3)

end

--OR-

for k, v in pairs(lantern) do

v:SetPosition(-10 + k, -3) ←- this is probably faster

end

The pairs loop ensures that you will hit every object in the named group, and the value of k for the current iteration will be the specific actors index that it was labeled with. Honestly, I was going to write a section on the disadvantages of using indexed names, but the pairs loop pretty much fixes everything. Oh, also:

\_G[“lantern[2]”] ←---------------------BAD MONKEY

GetActorRef(“lantern[2]”) ←---------------------do this

Also, I’m impressed if you’re still reading at this point. Most people would be assembling their sniper rifle because of that error joke.

**Renaming and Deletion [RENAMING IS DEPRECIATED]**

If one of your actors ends up ~~in trouble with the mob~~ needing to be re-named for whatever reason, you can set its name to something else simply by calling SetName() on it and passing in your new name. You can even remove the actors name and reference entirely by just setting it to “” (open and close quotes). Everything will be taken care of.

Of it so happens that the ~~mob catches up with him~~ actor is deleted, we can remove the actor from existence entirely by calling its Kill() function. This function will cause the actors name to become nil, it will have its alive flag set to false and its engine resources will be released.

An actor can delete itself by using the rather darkly named:

self:Kill() -- =/

That was not intentional, I swear.

**Quirks with Deletion**

Deletion of actors is a tricky thing. Many engines don’t even allow deletion of objects at runtime because it causes so many complications. The problem with it is that you are destroying an object in a chaotic environment when there could be references to is still active. Deleting an object in that environment can easily cause show-stopping errors, and it can be very hard to craft systems that respond gracefully to that.

My general strategy was to heavily control where actors actually could be deleted. When you use the Kill() function, the actor is not deleted immediately. It *will* be deleted on the next engine update, which happens after the actor update pass of every frame (the loop that calls every actors Update() function).

I have thoroughly tested actor deletions, but there still may be subtle bugs in the system, especially if the actors are being referenced by other actors (at runtime or in save files). As a result, be careful with deletion and try to be sparing with it. I would say, as a rule of thumb, don’t delete anything that ever will be referenced by something (other than a simple piece of one-time code) and prefer to have actors delete themselves (usually via their update function and some condition being fulfilled).

**Tiles**

We now move from actors, to a special case of actors that we, somewhat inappropriately, call tiles. Tiles are a general actor with several pre-set configurations that make it very easy to define new sub-types of them. So easy in fact, that you can declare one, or even multiple in a single line of code, but we’ll get to that later.

Tiles are generally described to be static and do not directly participate in gameplay. That is, they do not move on their own accord and are not referenced by animations, events or other actors. This makes them possibly the simplest type of standard actor in the script system.

**Passive Tiles**

I’ve talked about passive tiles before, and they’re pretty basic. They have a single object with a sprite subatom and... that’s it. Given that they are actors, and they do own a generic sprite object, they can theoretically be animated, but they are not streamlined to do this. There’s really not too much to say here. They’re defined by a transformation and a sprite, with potentially all the properties of a sprite included with that (of course).

Surface tiles are normally used for backgrounds, fine-detail decorations or “fill tiles” to cover up large areas of open space.

**Surface Tiles**

Surface tiles are actually quite complicated beasts. They include all subatom types except for the character subatom. This is because, as their name implies, they compose surfaces of the terrain and world. Surfaces have to interact with objects, projectiles, lights, pretty much everything.

They are treated the same as passive tiles in the world editor and as far as save files are concerned, but they have a few extra parameters to worry about involving physical properties. They also have an extra class defined on them that we will cover in a second.

**Passive Tile Leaf Classes**

We can create an enormous palette of tiles with relatively few instructions with the leaf class concept. Thanks to lua being a dynamic language, we can write a series of function that can automatically create new tile classes for us. We can say that a **leaf class** is a **specific configuration of a general tile class**.

A specific configuration of a passive tile is really nothing more than the definition of an image to use and a pattern scale. The image to use is defined by a filename and a pattern scale is “how large” one repeating tile of a texture should be on screen.

Creating and placing these in the world is easy, and we will cover that in a bit.

**Surface Tile Behavioral Classes**

Unlike with passive tiles, it’s not particularly useful to define new types of surface tiles with an image and a pattern scale. Surface tiles need to interact with things, and to do this they need a layer on top of their general class that lets us define their specific behaviors and how to use their subatoms to respond to stimulation from the environment. This really boils down to simulation of different materials in the environment. I would call them Surface Tile Material Classes if I wasn’t worried that name may limit the thinking patterns of the scripters to come up with inventive types of surfaces.

These “behavioral classes” inherit from the general surface tile class and consist of a series of functions that we can use to define the behavior of the tiles. This mostly amounts to checking input from their subatoms to see if anything has hit them or sent them an event of some sort.

The main way that they will respond is through their update function. They can do things such as check their projectile subatoms for impacts, their physics subatoms for collisions and their event subatoms for events broadcasted near them. With this, there’s lots of ways to make surface tiles respond in interesting ways.

We could, for instance, have a metal surface behavior class. We could check its projectile subatom and detect impacts, causing a spark or even a ricochet (by cloning the bullet with the tiles own subatom and replicating the data). We could also check the tiles physics subatom and detect hits that have happened. If the hits happen at steep angles or slide with a heavy force, we could cause a stream of sparks to flow from the collision point and play a screeching metal sound with its sound emitter subatom.

So, in essence, the behavioral class is just a custom update function.

[NOTE: this part of the system is currently unfinished and creating new behavioral classes is a messy process. I would leave this alone for now.]

**Surface Tile Leaf Classes**

Unlike passive tiles, you cannot directly define leaf classes of surface tiles. Instead, you need to define leaf classes of their behavioral types. As far as defining leaf tiles for behavioral classes though, it’s very similar to defining them for passive tiles. It can be done with a single function call that specifies an image name and various properties. Along with the pattern scale, surface tiles also need a shape specified on them, as well as a horizontal flip parameter for the geometry.

[NOTE: at the moment, the only shapes avalible are a box and a ramp that goes up at a 30 degree angle. There will be more in the future.]