An Entity-Component model for Extensible Virtual Worlds

Abstract

We propose a model with basic building blocks on which arbitrary virtual worlds applications can be built. It consists of Entities which can have any set of Components attached, and application developers can define own components to add arbitrary data that they need for the specific functionality. The components are basically just data stored in the Attributes of the component. The values of the attributes can be automatically synchronized among the participants in the networked environment.

Additionally we present the idea of Entity Actions. ...
http://groups.google.com/group/realxtend-dev/browse_thread/thread/42fed16befd2b9b7/bba9f67d00b60371?lnk=gst&q=

Contents

Introduction

A virtual world can be potentially anything. An anatomical simulation of the internals of a biological body, where you can fly around inspecting the functionality of various organs and blood vessels etc. has certain kind of entities, and astrological simulations or solar systems and galaxies has quite different ones. A game of hockey has the concepts of attack, neutral and defending zones, certain conventions for which kind of physical interactions between the humans in the game are allowed, and of course the notions of a goal and a puck. Figure skating has none of these, even though the physics are identical -- a skating ring with people skating. The user interface controls and functionality in those two skating games are largely different, for example in the team based hockey you want good controls for passing the puck to team mates, whereas in figure skating it might be something like elaborate controls for the jumps and the skating itself, and higher level selections for how to proceed in the pre-planned choreography of the performance.

(All these cases are easily virtual worlds, without streching the definition. We could go even further to define abstract worlds with arbitrary behaviours, like a "world" where in one part there is a space where text characters can exist, and another area where there are buttons with actions like load and save. A text editor application, that is. In fact, one definition of a computer is that it is a virtual world generator [Deutch]_, so following that perhaps would make all software virtual worlds. This line of thinking might not be fruitful, but the extreme genericity is something to concider still. For example in games not all geometries are euclidian, perhaps VW archs should not hardcode that.)

What should a generic virtual world architecture provide to facilitate the implementation of all those, and many quite different, applications? It seems we need both arbitrary custom data and functionality. For example in the astronomical simulation the data would be mass and heat and other properties of the planets and suns. In a functional anatomy simulation you might need chemical information (...). To be able the fluently control and navigate these vastly different applications, specific user interface controls would be useful. In the skating games, the custom user interface actions would need to be tightly coupled with how the character animations (e.g. a slapshot or a (kolmoishienohyppy)) are executed. All have common needs too: need to represent visual 3d objects, move and animate them, and to synchronize all the data among the participants and to store it for continuing use.

We propose an architecture that meets these requirements using three main elements:

1) A scene model based on the notion of a generic entity, to which any kind of components which can contain arbitrary data can be added.

2) A core API for basic functionality like showing graphics and getting user input, and manipulating the scene to e.g. create new entities. And a module and scripting system that can be used to add arbitrary code that uses the core API to be executed in all parts of the system (e.g. the client and server).

3) An extensible network protocol suitable for real time use, which allows applications to define own custom messages.

This has been implemented within the open source realXtend project next generation architecture effort, resulting in a generic virtual world software framework in the Naali application. Naali was first developed as a viewer client application, to replace the previous realXtend viewer prototype which was based on the Linden labs Second Life (tm) viewer. The server counterpart is Opensimulator (or Opensim), a modular open source application for generic virtual worlds which features a SL (tm) compatible server implementation out of the box but targets supporting other kinds too. The original realXtend functionality(*) on the server side was already implemented as an Opensim module (ModRex), and we have now added the support for arbitrary scene data using the Entity-Component model there too. The opensim+modrex combination, the counterpart of Naali(*) is called Taiga. Naali is written in C++ and uses the Ogre3d graphics rendering engine and the Qt toolkit for UI and internal architecture like supporting dynamic languages (currently Python and Javascript, both as optional modules, and more can be added in additional modules).

(* original realXtend functionality, first published in February 2008: Ogre meshes with animations and complex materials, mediaurls ('web pages as textures'), ..)

(* Naali is the Finnish name for the arctic fox -- Naali strives to be a generic 3d web client app, the Firefox of virtual worlds)

Now (since October 2010) Naali has also the capability to create a scene by itself and optionally run a simple server module, so it can run standalone also without Opensim and provides an alternative server implementation using a different, more efficient and featureful extensible protocol than the LLUDP provided by current Opensim. This new protocol & server experiment is called Tundra, as an alternative for the Opensim based Taiga, and it uses a new networking library called kNet (or Kristalli) which is similar to eNet but with certain advantages. If the Tundra experiment proves succesfull, and there are use cases that need both the kNet protocol and Opensim features, one possibility is to add a kNet client stack to Opensim. The Naali client works against both server implementations, and for application code that e.g. defines new Entity-Components and uses existing ones, the protocol doesn't show as the data is gotten and manipulated with the same protocol agnostic API. Tundra uses ECs extensively to define everything -- there is nothing hardcoded about what a world likes, unlike with LLUDP where the Second Life (tm) application is hardcoded: every user has an avatar, there is a single certain kind of terrain etc. With the new generic realXtend architecture demonstrated with Tundra that actually becomes data: a description of a certain kind of a scene with default entities.

This article is structured as follows:

First there is an overview of related work. This covers analyses of pre-existing virtual worlds solutions like Second Life and the vanilla Opensimulator server and the original realXtend prototype made using those. Then an overview of literature about solving the extensibility problem, suggesting the Entity-Component model, based on which we selected that for the new realXtend architecture. We also take a look at other related systems utilising ECs and custom networking, especially the proprietary Unity3d game engine and an open source virtual worlds framework called Syntensity.

Then we present the realXtend architecture in more detail, the designs of the three main elements: the extensible Entity-Component scene model, the core API and the extensible network protocol. Using these to make custom applications is then illustrated with simple highlevel descriptions of some examples. One of the examples is a reproducing the basics of the Second Life (tm) like featureset using the generic API. Other examples feature different worlds, ones without avatars nor land nor controls to move single entities around.

Chapter X is a brief overview of the implementation, covering the main technologies used in Naali: Ogre, Qt and kNet. How these were used to implement the ECs and the core API. (also the module system etc. of course).

(something more still? then discussion/evaluation, future roadmap, conclusion)

Related work

This covers analyses of pre-existing virtual worlds solutions like Second Life and the vanilla Opensimulator server and the original realXtend prototype made using those. Then an overview of literature about solving the extensibility problem, suggesting the Entity-Component model, based on which we selected that for the new realXtend architecture. We also take a look at other related systems utilising ECs and custom networking, especially the proprietary Unity3d game engine and an open source virtual worlds framework called Syntensity.

Architecture design

1) A scene model based on the notion of a generic entity, to which any kind of components which can contain arbitrary data can be added.

2) A core API for basic functionality like showing graphics and getting user input, and manipulating the scene to e.g. create new entities. And a module and scripting system that can be used to add arbitrary code that uses the core API to be executed in all parts of the system (e.g. the client and server).

3) An extensible network protocol suitable for real time use, which allows applications to define own custom messages. http://clb.demon.fi/knet/index.html

The key aspect of extensible virtual world architecture design is to allow new features and functionality to be added to existing world objects, or modifying the existing behavior without the need to modify other parts of the system. In Naali means of extensibility are provided by the concepts of modules, the scene-entity-component-attribute model and extensive support for dynamic scripting languages.

Naali Framework and Core Application Programming Interface

Framework

XXXjohonkin sopivaan väliin: Same codebase on both client and server. The root object for accessing all Naali features is the Framework object. This object drives the main loop and gives access to core application programming interfaces (API) for modules,entity-compoennts and scripts.

The communication between framework, its APIs, modules and scene entities is implemented mainly by using signal and slot mechanism provided by Qt. A signal

> is emitted when a particular event occurs and a slot is a function that is called in response to a particular signal. For example, if developer is interested in listening input events, he registers input context from the input API and connects input context■s KeyEvent() signal to his HandleKeyEvent() function. Now every time that keyboard input event is received in the system, input context emits the corresponding signal which can be then handled by the user as wanted.

The functionality built into Naali is subdivided into core libraries and optional subsystems implemented as sets of modules. The core libraries are sets of APIs that each expose some functionality of the Naali system to native C++ code, scripted modules or scene entities. These interfaces are accessible directly through the root Framework object. The individual subsystems are accessible by querying the appropriate modules using the Module API. The subsystems include functionalities such as renderer, voice and instant messaging communication and scripting support.

The Naali SDK consists of 12 ■ or 13, if Naali is acting as a network server - core APIs: -Module: gives access to all the modules registered in the system. -Event: channel for system-wide inter-module messaging.

> Config: implements a uniform mechanism for modules to store their configuration data.

-Debug: contains useful development-time mechanisms like memory tracking and execution time profiling.

-Input: provides access to Qt input devices in a contextual order. -UI: exposes the Naali main window and allows clients to create and show their own

> Qt-based windows. painted on the same canvas as the 3D world is rendered.

-Scene: gives access to the Naali 3D scene structure: scenes, entities, components and attributes.

-Asset: implements the Naali asset system and a pluggable asset provider architecture for streaming in asset data files. The Asset API is uniform regardless of the underlying asset provider.

-Console: allows modules to register new console commands that are invokable from a command line console.

-Connection: exposes the functionality related to the currently active server connection, if Naali is connected to a server.

-Audio: provides functionality for playing back 3D positional audio and non-positional audio, adjusting master volume for different categories of sound, as well as recording sound.

-Frame: exposes signals for frame- and time-related event processing. -Server: if Naali is acting as a network server, this interface can be accessed to interact

> with the currently active client connections

Module

For creating a cross-platform pluggable framework Naali utilizes shared, dynamically loaded libraries. The libraries can be distributed separately and as long as interfaces stay stable, and also can be used with different versions of the framework without the need to update the library itself. The optional Python script module provides also its own mechanism for creating new modules, which can be loaded either locally or from the web. The semantic of module and component varies greatly in the field the computer science, but in this article module refers to a plug-in module created for Naali system to add new functionality, and component refers to an entity-component (EC) in the scene.

XXXModule load, init, unit jne. tsydeemit lyhyesti?

The Scene-Entity-Component-Attribute Model

The scene-entity-component-attribute model describes the extensible virtual world.

Scene

At the highest level, the world is composed of one or more scenes. Scenes are generic containers for entities and are owned by the main application framework. Scenes are identified by their name. Scenes can be accessed, created and removed through the framework.

Entity

An entity is a unique presence in the scene identified by a unique ID number. Entity itself doesn■t contain application-specific functionality or data and it is a mere generic container of components, which define data and behavior for the entity. Entities can contain any number of components of any type, even several different instances of the same type of component. It is up to the component to make sure this causes no problems. Entities can be local (non-replicable) or networked (replicable). Name, description and other metadata can be added by using components. Additionally entities can be temporary, which means they won■t be saved when the scene is. Entities are created, accessed, and deleted through the scene.

Component

Components, also referred as an entity-components (EC), are containers of entity-specific data and behavior. Components are identified by their type name and name. In the original design, in early 2009, components were designed to be very lightweight data containers and minimize the amount of behavior they define and let parent modules handle the behavior, based on the data the component contains. However, as time passed by, it was found more logical and sane to contain also more functionality within the EC. For example a light EC has a private pointer to an OGRE light entity. Every time user changes the EC_Light■s color attribute, the value of the attribute is applied to the OGRE light resource internally by the EC. Existing components are accessed, created and deleted through entities, but it■s also possible to create new components through framework■s component manager. The network synchronization of specific component can be defined separately, but by default all components are synchronized.

Attribute

Attributes are containers for the data of the component. Component can contain any number of attributes. An attribute has a specific type, for example integer, real number, string, or a 3D vector, and a name. An attribute can contain optional metadata, such as minimum and maximum value for the attribute. When setting new value for attribute, the type of change has four options: -Default: uses the current synchronization method specified in the component this

attribute is part of.

-Disconnected: The value will be changed, but no notifications will be sent
(even locally). This is useful when doing batch updates of several attributes at a time and wanting to minimize the amount of re-processing that will be done.

-LocalOnly: The value change will be signaled locally immediately after the change

occurs, but it is not sent to the network.

-Replicate: After changing the value, the change will be signaled locally and this change is transmitted to the network as well.

Entity Action

Additionally we present the concept of entity action (EA). EAs allow more complicated in-world logic be built in slightly more data-driven fashion. The idea is to augment the entity and component with the concept of actions. EA is identified with a string, for example "Open", "Talk", or "ToggleVisibility". An action can have any number of parameters, for example: "Move(up)■, ■ WalkForward(10)■, ■StartGame(level2.lvl, listOfPlayers])". The reason why we are using strings is to keep the interface as simple as possible. More complicated data passing can be achieved by first filling in a series of component data and then calling an action that triggers the event.

It sounds more natural to have explicit actions that can be performed on entities, instead of trying to achieve the same by listening on edge-triggers on EC data changes. I.e., instead of triggering the locking of a door by listening in a script when a component's "locked" transitions from false to true, we can do an explicit "Lock" action, which then locally sets "locked=true" and e.g. plays an audio clip.

As a lesser note, this also avoids us having to transmit over the wire lots of data in several situations. E.g. we want to clear a component to its default values. Instead of synching all ECs individually, we can execute an action "Reset" on the entity, and locally we know which values to fill in.

Entities have always been designed to be simple objects and the Entity class is not subclassable. The Entity class itself will not handle any actions. Instead, the actions are passed on to the components of that entity. By adding the concept of Action to the entity, instead of individual components we do not have to know about the actual components the entity has, which allows all sorts of "hidden layers of abstraction", i.e. we can trigger an action "Clicked" on an Entity from the input system, without requiring the input system to know which component is to handle the action.

Also, it allows a layer of consistency. We can make sets of components that behave similarly. E.g. components Mesh, ParticleSystem, Billboard, Avata and Light all have in common that they have the notion of actions "Hide", "Show", "ToggleVisibility". A piece of code that needed to know about the components would have to do a chain of if-else-condisitions to see which component was there and which had to be hidden.

Entity will get a signal table of actions. A component can register as a listener to an EA. This makes the entity actions dynamic and run-time addable and removable from script, instead of being static compile-time declarations. For debugging, and for more power in the UI editor, the dynamic entity action signal map can be made accessible from the UI.

XXXremove start? We need to be able to know whether an action succeeded or not, or if anything even handle the action we sent. These are discussed separately below. Letting the handler of an action signal success or failure, or other pieces of information, is a more complicated question. Success or failure can be expressed by using exceptions. Implementing "Action Return Values" would make the actions real function calls, like "Entity.Exec("GetName")". This topic sounds somewhat interesting, but is for later study since there are complications, one example below. If multiple handlers are connected to an action, which one gets to handle it? Do we implement a similar kind of suppression mechanism as with Naali events (return "Suppressed"/"NotHandled";)? In which order do the handlers get to the action? If we have Action Return Values, do we do a "yield return"-fashioned aggregation if all handlers return a separate value? We recommend to leave the execution order undefined, and to pass the action to all handlers without the ability to suppress. XXXremove end?

When we execute an EA, for example "WalkForward■ or "UnlockDoor", where should the action be executed? There are three possible places that can be immediately identified: local client, server and peers. And as combinations we get local + server, local + peers (all clients but not server), server + peers (everyone but me), local + server + peers (everyone). Not all of these sound immediately sensible even, but we know we need to be able to do different things at different times. This way, server-side execution can be seen as a kind of remote procedure call (RPC) method (without return values currently). Since in the Tundra branch our client and server maintain the exact same scene data structure, we can also utilize

the same scripts, and changing between client and server execution is fast for prototyping.

The server execution works as follows. In a client-side script or native C++ code that obtains the handle of an Entity, we can call entity.Exec("Unlock", Server). This will not do anything immediately on the client, but will cause the action to be serialized through network and triggered in the scene on the server side. This way the server script can check and enforce that the caller is actually permissible to do such thing, and act properly. The client is not able to unlock the door with local execution alone.

In the protocol, an "EntityAction" network message is implemented that carries the entity, action and the corresponding parameters. This way most of the client-server interaction scripts do not need to implement their own network messages at the bottom protocol layer for custom messaging, although, for more power, having also such an option is useful.

XXXremove start? How do we know which actions are available on the server if it differs from the local computer? An idea might be to have an ■ActionQuery■ message, but this is not critical the time being. Perhaps we should prefer to make symmetric client and server -side action registrations to keep it simple. XXXremove end?

When executing over the network, the handlers need to be able to know the originator of the action, i.e. whether it was local or from which peer. The individual security checks are left to the scripts themselves to decide at this point.

Real-time Network Synchronization

An extensible network protocol suitable for real time use, which allows applications to define own custom messages.

Immediate mode and XML stuff.

By default changes in component data, i.e. attributes, are synchronized on a per-attribute basis. Batch updates of several attributes, or even whole component, can be done using the Disconnected change type and triggering the synchronization signal by setting the value of the last-to-be-changed attribute with the Replicate change type.

Per attribute vs. per component sync?

Scripting

In order to support agile application development and fast prototyping with dynamic scripting languages, exposing the C++ written functionality for the scripting languages is crucial. In Naali every core API and scene model -related objects are exposed to both Python and Javascript allowing usage of scripting languages to be almost as powerful as native C++ code. The script supported is not limited only to the two aforementioned languages, and implementing support for example LUA is very possible if wanted.

Functionality example

To showcase extensibility of Naali in action, we will go through implementations of two example applications. The first one shows us how to implement a Second Life -like concept of avatars. The second example will show us how create a virtual world without the concept of avatars at all by implementing a 3D presentation browser.

By default new scene is a black and empty void. So for starters in both examples we have created without separate annotation an entity which has an environment component so that we can set the lighting for the scene.

Avatar

The avatar application scene implements the concept of avatars. It uses two script files: application script, which handles user connections and creates and deletes avatar entity as users leave and join the world, and a avatar script, which implements the functionality ■ movement, animation and physical behavior - of a single avatar. For resources, in addition to the script files, we have a scene XML file, which describes the scene entities and components in the scene, a mesh, skeleton, material and textures for the avatar and a mesh for the ground entity. The application script itself is a very simple server-side script. Every time new user connects to the server, server creates a new entity for the user and sets EC_Script, EC_Placeable and EC_AnimationController components to it. Server also sets random starting position for the avatar in the world. When user disconnects from the world the application script will destroy user's entity.

Avatar script blaablaa

The avatar entity consists of the following ECs: -Mesh: assigns 3D mesh object and skeleton to the entity. -RigidBody: applies physics to the 3D object. -Placeable: postion, rotation and scale of the 3D object. -InputMapper: nonsynced input-mapper registers an input context from the Naali input

> system and uses it to translate given set of key and mouse sequences to Entity Actions on the entity the component is part of. InputMapper maps the keyboard events to entity actions (server execution type) which are sent to server.

-AnimationController: controls the animation states of the skeleton of the mesh. -Camera: implements camera to the scene (a third person which follows avatar from behind). -Name: used for setting name and description for the entity (optional).

Creates camera and input mapper on client. Client controls camera. Avatar created on server. Connects movement and mouse look related actions. Server receives movement actions and updates the position and animation, runs physics simulation server handles animation state changes, client playbacks current state's animation. // If walk animation is playing, adjust its speed according to the avatar rigidbody velocity // Note: on client the rigidbody does not exist, so the velocity is only a replicated attribute Server: ServerInitialize() ServerUpdate() ServerUpdatePhysics() ServerHandleMove() ServerHandleStop() ServerHandleRotate() ServerHandleStopRotate() ServerSetAnimationState() ServerHandleMouseLookX() ServerSetAnimationState() Client: ClientInitialize() ClientCreateInputMapper() ClientCreateAvatarCamera() ClientUpdateAvatarCamera() Common: CommonUpdateAnimation()

Presentation

(the way I started writing the pres example ended so long that putting it to a separate page now, [[Platform Extensibility Working Group/Presentation Application]]

The Naali implementation

a brief overview of the implementation, covering the main technologies used in Naali: Ogre, Qt and kNet. How these were used to implement the ECs and the core API. (also the module system etc. of course).

The main libraries used to implement Naali are Qt, OGRE and KristalliNet.

Qt is a powerful, visually rich, cross-platform, C++ graphical user interface application programming framework. It supports all major desktop OS, as well as many embedded platforms. Currently Naali has been built and run successfully on Windows XP/Vista/7, couple of major Linux distributions (Ubuntu and Fedora), OS X, Maemo and MeeGo.

OGRE (Object-oriented Graphics Rendering Engine) is an open source and cross-platform full-featured 3D rendering engine. A common misconception about OGRE is that it is a game engine. OGRE is a mere 3D rendering engine and it is missing many essential game engine features, such as sound, input and networking.

XXX suora copy-paste http://clb.demon.fi/knet/ : KristalliNet (kNet) is a connection-oriented network protocol for transmitting arbitrary application-specific messages between network hosts. It is designed

primarily for applications that require a method for rapid space-efficient real-time communication. KristalliNet consists of two major specifications. At a lower level, the Transport-level specification defines methods like: -How a session is initialized, maintained and shut down. -How data bytes are transferred between hosts. -How reliable messages, data ordering requirements and large transfers are handled. -How flow management and congestion control may be performed.

Building on top of that, the Application-level specification, along with a reference C++ implementation, offers most notably the following features: -How to describe your own application-level protocol sets and their transmission parameters. -How to model message dependencies and prioritization. -How to perform content-based late data replacement. -How to perform data serialization.

Declarative (XML) http://clb.demon.fi/knet/_kristalli_x_m_l.html and immediate.

XXX PythonQt and Qt Script Generator provide Qt-related bindings for Python and Javascript.

XXX OpenAL for audio.

Naali Software Development Kit (SDK)

Core Components Naali provides a set of built-in entity components, which are the building blocks that make up the world. New arbitrary components can be defined in C++ by writing a new Naali module to host it.

XXXjottain järkevämpää tähän Name, Environment, Placeable, Mesh, InputMapper, Movable, AnimationController, ParticleSystem, Sound, Listener, Camera, SoundListener, Light, HoveringText, Billboard, DynamicComponent, Script, SkyBox/SkyPlane/SkyDome (singleton), Terrain, WaterPlane, Fog

To allow dynamic behavior for data driven ECA data and script contents, the following special components exist: -DynamicComponent: Unlike other components, the attributes of this component are

> dynamically defined at runtime and can be specified per instance. This is particularly useful for scripts for managing their own custom application-level data.

-Script: Attaches a script source file to an entity, implemented either in JavaScript or Python. This allows attaching custom application-level logic to an entity.

Discussion: evaluation, future work

Conclusion

References

- A unified component framework for dynamically extensible virtual environments, http://portal.acm.org/citation.cfm?id=571878.571889&type=series
- This seems like one to check for the article, and open source project by some uni with publications,

http://oldsite.vrjuggler.org/papers.php has things like "VR Juggler -- An Open Source Platform for Virtual Reality Applications" latest release is from march 2008, i guess the project is still living. the full proceedings of IEEE VR from the past couple of years would be the thing to check .. that one was there in 2001

- the bimonthly presence journal seems relevant, had this framework thing in 2003 .. has been cited this year too: http://www.mitpressjournals.org/doi/abs/10.1162/105474603763835314