# Virtual Theater Design Document

## Table of Contents:

# Teams

| | |
|---|---|
| **Audio:** | Jordan Caras |
| **Camera:** | Paul Solt |
| **Events:** | Rohan Mehalwal<br>Kyle Tirak |
| **Grabbing:** | Nicholas Wilsey<br>Robert Yates |
| **GUI:** | Colin Doody |
| **Luster Mocap Plugin:** | Andrew Galante<br>Abhishek Moothedath<br>Rodrigo Urra |
| **Mocap Suit:** | Erik Carlson<br>Yuqiong Wang<br>Pranabesh Sinha |
| **Models and Lighting:** | Colin Doody<br>Michael Romero |
| **MotionBuilder Plugin:** | Abhijit Bhelande<br>Andrew Brown<br>Chris Murdock |
| **Networking:** | Jordan Caras<br>Will Jennes<br>Stacy Pine |
| **Object Highlighting:** | Dan Wisnewski |
| **Overall Framework:** | Colin Doody<br>Michael Romero |
| **Oversight:** | Michael Romero |
| **XML Parsing:** | Colin Doody |

# Audio

## *Summary*

The goal of the audio was very simple- provide a single, simple interface for loading and playing any type of the 3 sounds: compiled, loaded, and streamed. Compiled sounds are the ones that are specified as a resource to the project. They must be known ahead of time and exported with the project. Loaded sounds are read in at run time via byte arrays- this happens asynchronously and took special handling. Streaming sounds are files specified by path (URL or local)- they are streamed rather than loaded into memory.

## *Design*

The SoundManager is responsible for handling all Luster sound objects. The manager maintains a list of internal Sound objects. Because sounds come in three types, each type is its own subclass and has its own way of loading, or playing, or both. With each subclass handling itself, the manager needs only to treat each of its sound files as a Sound object.

The limitations of the different types of Luster sounds drove the design of the audio system. Luster has 2 types of sound objects: a Luster Sound and a Luster Instance. For all sound types, an instance can only be played once. These instances are created from the Luster Sound. We wrap this object into the Soundlet class to give it more properties.

Multiple instances can be created from a single sound and play simultaneously, which is quite easy. Luster Streaming sounds, though, complicate things. Only a single instance created from a single sound can be playing at a time. This implies that for every stream play, a new sound must also be created. Since this is an expensive operation, the StreamingSound class implements its own pooling mechanism.

The last complication involves the Luster Sounds which are loaded at runtime. Because this is done asynchronously, all sounds act as EventDispatchers, alerting the manager when they are loaded; the synchronous sounds alert immediately for consistency.

## *Functionality*

Any sound can be played any number of times simultaneously.

## *Usage*

The outside world should only be concerned with the SoundManager.lua class. This class provides methods to add(), create(), and play() sounds. Create will batch load any unloaded sounds, so call once, or many times. The SoundManager class also acts as an event dispatcher, dispatching SOUNDS_LOADED whenever the last sound since the last create call finishes loading.

## *Luster Issues + Limitations*

Cannot play from an arbitrary position in the sound file.

## *Further work*

### VOIP

Voice over IP would be a great addition to the sounds, especially if it could be implemented in a seamless sound playing sort of way. A plug-in would need to be developed.

### Direct Sound Access

Right now, play returns a unique identifier upon playing a sound. The reason for this would be to call pause( id ) on the SoundManager to pause a given play. This approach is not the best, as it requires a larger call stack just to access a Soundlet. A better way would be to return a limited functionality sound object up the stack what has properties like pause(), duration(), timeleft(), etc. Since each play uses its own sound instance, the instances are not pooled, so this is a possibility.

# Camera

## *Summary:*

The camera management in the virtual theater project allows you to add cameras to view objects and move around a scene. There is support for multiple cameras and there are different modes for the camera: free look, tracking, and locked. The different modes respond differently to the camera input.

## *Functionality and Usage:*

**Classes:** CameraManager.lua (vt.CameraManager), Camera.lua (vt.Camera)
**Camera Input:**

- WASD/Arrow Keys will move the camera.
- Left-click and drag will rotate the camera.
- Z/X will lower and raise the camera.

Usage:

```
-- Create or use an existing scene
self.scene = luster.display.Scene()
-- Create the CameraManager and set the scene to view
self.cameraManager = vt.CameraManager.Instance( self.scene )
-- Add a full viewport camera called "Camera1"
self.cameraManager:addCamera("Camera1", vt.Camera.ONE_VIEW)
-- Set the focus for the input to "Camera1"
self.cameraManager:setFocus("Camera1")
```

The CameraManager will attach new cameras to the scene. Focus is set to a specific camera, in this case the only camera, so that input is correctly sent to the camera. With multiple cameras viewing a scene there should be at least one camera with focus.

**Camera View Modes:**

Camera.ONE_VIEW - The camera uses the full screen
Camera.FOUR_VIEW_TOP_LEFT - The camera is in the top left corner
Camera.FOUR_VIEW_TOP_RIGHT - The camera is the top right corner
Camera.FOUR_VIEW_BOTTOM_RIGHT - The camera is the bottom right corner
Camera.FOUR_VIEW_BOTTOM_LEFT - The camera is the bottom left corner
Camera.TWO_VIEW_TOP - The camera is the top half
Camera.TWO_VIEW_BOTTOM - The camera is the bottom half
Camera.TWO_VIEW_LEFT - The camera is the left half
Camera.TWO_VIEW_RIGHT - The camera is the right half

**Camera Modes:**

Camera.FREE_LOOK – Input can be used to reorient the camera's direction and position.

Camera.TRACKING – The camera is set to tracking when asked to track a node in the scene. Do not set directly with current system.

Camera.LOCKED – The camera cannot move its position or orientation

Camera.POSITION_LOCKED – The camera cannot move it's position, but can move its orientation.

Camera.ORIENTATION_LOOKED – The camera cannot move its orientation, but can move its position.

## *Luster Bugs and Issues:*

**Bug 1:**

The stage.root variable needs to be set, or there will be issues attaching the CameraManager to a new project. For example in a new luster project in the Root.lua file the following line needs to be set. Errors may be thrown if not set. This line will initialize the root scene stored in the luster stage class to be the current scene, which is used in some callback functions.

stage.root = self

In the VT application, the stage.root is set in the SceneManager as a workaround for the bug. There seems to be something wrong with luster stage or with the order of initializations.

**Bug 2:**

The getPitch(), getYaw(), getRoll() functions do not work in luster.Quaternion, it crashes in the C function call. I created work around functions in the Camera class to get these attributes.

**Issue 1:**

There is no support in Luster to interpolate between quaternions. This makes it hard to get the orientation of the camera to smoothly move between objects. There should be other helper functions to deal with quaternions which can be found in Ogre3D.

## *Future Work:*

**Smooth rotations** – There needs to be support for smooth rotations between objects when switching tracking. Currently the camera will abruptly change focus and this is not ideal for a viewer. There is no interpolation between quaternions in Luster. I created a simple linear interpolation function. However it doesn't use the shortest arc for movement in all orientation transitions and can take the long arc around. It should be a spherical linear interpolation rather than a linear to keep a constant rotation.

**Look at directions** – The camera needs to be more flexible and let the user look at certain objects without disrupting up the free look mode. I worked on this a bit, but was set back by various issues regarding quaternions support in luster.

**XML Description** – The camera needs to be able to be scripted by cues in XML. This involves modifying the CueManager and the XMLParser to add support for positioning and orienting a camera. This also relies on having functions to move/transition the camera similar to how cues for lights can be cued in the theater project.

**State Switching –** There should be support to switch states and have the camera system correctly handle the switches. Most of the state functionality is complete, but needs to be glued together.

**Rails** – It would be good to have rails that constrict a particular camera to a line or arc of movement. Input would need to be remapped accordingly to the rail.

# Events

## *Functionality and Use*

The events in the Virtual Theater project have been implemented mostly in Cue.lua and CueManager.lua in the VT_Core project. The cues are specified within an external XML document for a particular scene. The XML parser is then responsible for creation of the cues for use by the system internally.

There are 4 types of cues currently implemented, the design is such that it should not be overly difficult to add a new type of cue should a need arise. The 4 cue types are "light", "model", "animation", and "audio". Each has its own set of possible actions associated with it. The light and model events will modify the light or the model's state in the theater space. "Tweening" is used for these types of events to have smooth lighting and model transitions and transformations.

As per the design, Cues in the Vitual Theater project are a set of events that will happen when the cue is executed. Each event has a delay attribute that will have the event wait a certain number of milliseconds after the cue is executed before the event begins. Event chaining is possible in this way to allow for very accurate synchronisation of sound, actions, and lighting once a cue is executed by a client.

Client access for the cues is also specified in the XML file. This can be used to ensure that only specific clients can execute certain cues. For example, if only the pupeteer is desired to be able to execute an a cue associated with a character, that cue would enable access for the pupeteer client and for nothing else.

## *Luster Issues*

The only outstanding Luster issue encountered was the difficulty involved in tweening orientation for light (spotlights) and model events. Since orientation in Luster uses quaternions, it is difficult to tween this attribute.

## *Future Work*

Orientation tweening should be added at some point. It would also be a nice to have a graphical editor for cues that could write the necessary cue information into the XML file from the stage manager client.

## *Cue XML Structure*

Cues for a scene are placed within the <CUES> which is placed within the <SCENE> tag in the XML scene description. It should look like the following:

```
<SCENE>
	<CUES>
		(one or more Cue specifications)
	</CUES>
</SCENE>
```

Each cue has the required attribute of a unique string ID that will be used to distinguish a particular cue for that scene. A cue must also be given the types of clients (stage manager, puppeteer, audience, and actor) that should be able to execute that particular cue. This XML represents a cue named "Act I Opening" that is able to be executed from any client.

```
<CUES>
	<CUE id="Act I Opening">
		<ACCESS client_type="Audience" />
		<ACCESS client_type="StageManager" />
		<ACCESS client_type="Puppeteer" />
		<ACCESS client_type="Actor" />
		(One or more events associated with this cue)
	</CUE>
	...
</CUES>
```

Now we are ready to add events to this cue. The <EVENT> tag is used for this. Each event in a cue is required to have a type, an object ID, a time, and a delay. The type specifies the type of event this is and determines how the event will be executed. Possible values for the type field are "light", "model", "animation", and "audio". The object ID specifies the entity within the scene that the event will be associated with. For light, model and animation events, this will be the string ID of the light or model being altered. For audio events, this will be the string ID of the sound resource. The time field is used to regulate the time taken for light or model events where "tweening" is involved. It will be the time it takes for the event to reach its end state in milliseconds. The time field is not used for sound or animation events, but still must be specified (you can just say 0). The delay is the time in milliseconds that will pass between when the cue is set off and when the event being specified will be started. The following will specify an event in the Act I Opening cue that affects the light MainSpotlight that will take 5 seconds to execute and will start as soon as the cue is set off.

```
<CUES>
	<CUE id="Act I Opening">
		<ACCESS client_type="Audience" />
		<ACCESS client_type="StageManager" />
		<ACCESS client_type="Puppeteer" />
```

```
            <ACCESS client_type="Actor" />
            <EVENT type="light" objectid="MainSpotlight" time="5000" delay="0">
                  (Event Actions)
            </EVENT>
            ...
      </CUE>
      ...
</CUES>
```

That event actions specify what is happening to the object associated with the event.

For light events, the possible actions are changing the color/brightness, moving the light, and changing the orientation of the light (direction for spotlights). For model events, the possible actions are moving the model, changing the orientation of the model, and scaling the size of the model. With the current exception of orientation, these values will be "tweened" based on the time attribute for the event (see above). These actions must have an end state specified. Start states for these actions are optional and will be used if present -- if not present the current value for the object attribute will be used as the start state. In this example, the MainSpotlight will changed to being off immediately and then tweened to fully on in the time of 200ms.

```
<EVENT type="light" objectid="MainSpotlight" time="200" delay="0">
      <START>
            <COLOR r="0" g="0" b="0" />
      </START>
      <END>
            <COLOR r="1" g="1" b="1" />
      </END>
</EVENT>
```

An example for changing the light from its current state to fully on would simply look like this:

```
<EVENT type="light" objectid="MainSpotlight" time="200" delay="0">
      <END>
            <COLOR r="1" g="1" b="1" />
      </END>
</EVENT>
```

Examples of other possible light event actions are shown here:

```
<EVENT type="light" objectid="MainSpotlight" time="200" delay="0">
      <END>
            <POSITION x="0" y="0" z="0" />
            <ORIENTATION yaw="0" pitch="0" roll="0" />
            <COLOR r="1" g="1" b="1" />
      </END>
</EVENT>
```

Here are some examples for other event types:

```
<EVENT type="model" objectid="Fridge" time="3000" delay="7000">
      <END>
            <POSITION x="0" y="0" z="0" />
            <ORIENTATION yaw="0" pitch="0" roll="0" />
            <SCALE x="1" y="1" z="1" />
      </END>
</EVENT>

<EVENT type="animation" objectid="Fridge" time="0" delay="7000">
      <ANIMATION name="Close" />
</EVENT>

<EVENT type="audio" objectid="moo" time="0" delay="0">
      <AUDIO name="play_moo" />
</EVENT>
```

Examples of scene specifications with full cue descriptions can be found in the CommonData/xml folder.

# Grabbing

## *Summary*

We added grabbing functionality to the Virtual Theatre application. When a grabable model is selected, the user chooses where on the actor to mount the model. Presently, the user can press either j or k keys on the keyboard to mount the object to either the actor's left or right hand respectively. The user can press the l key to unmount the object. In the future, mounting and unmounting could be done through context-menus. Right-clicking the mouse on a model would bring up a popup menu and the user could choose which mount point to mount the selected object to. Unmounting could be implemented by displaying a list of mounted objects and having the user select which object to unmount. Object highlighting and grabbing functionality are implemented in LUA scripts. The scripts call C++ functions in the mocap plugin to mount and unmount the internal Ogre entities.

## *Functionality + Usage*

To first enable grabbing on a model, one must edit the scene's XML document. The isGrabable flag on the model must set to true to make it selectable for grabbing. (Note: All grabable objects must also have the isPickable flag set to true in order to make the object selectable in the first place.) Using part of the example document below (VTScene01.xml), one can see how grabbing is enabled for the cube1 model:

**VTScene01.xml**

```
<MODEL id="cube1" src="smoothcube.mesh" isPickable="true"
isGrabable="true">
      …
<MODEL>
```

If the user moves the mouse over a grabable model, it then becomes highlighted and selected. At the moment, the keyboard must be used to mount and unmount objects from the Mia model.

Usage:
*Mount to left hand: Use j key
*Mount to right hand: Use k key
*Unmount: Use l key

In the future, the user should be able to right-click on the model to display a context menu. The user can then choose which body part on the Mia model to mount the selected model to.

The following code shows the keyboard function in VT_Client's Root file. It converts keys pressed by the user into mounting/unmounting functions.

**Root.lua (VT_Client)**

```
--------------
-- keyboard() handles keyboard events
-- Temporary Grabbing Code:
--    In the final version, mounting will be triggered from a context
menu.
--    From the menu, the user can select the mount point of
--     a particular model to mount to.
--    For now, mounting is triggered by key presses.
--                J/j:        mounts obj to Mia's left hand
--                K/k:        mounts obj to Mia's right hand
--                L/l:        unmount obj from Mia
--------------
function Root:keyboard(key)
      if key == 74 or key == 75 or key == 76 then
            local picked_model = self.sceneManager:getPicked()

            if picked_model ~= nil and picked_model.grabable then
                  -- for now, mount to a hard-coded model: id = "Mia"
                  local model = self.sceneManager:getModelByEntity
(self.sceneManager:getEntity("Mia"))

                  if key == 74 then -- mount to left hand
                        model:mountObject(picked_model, "Mia:LeftHand")
                  elseif key == 75 then -- mount to right hand
                        model:mountObject(picked_model,
"Mia:RightHand")
                  elseif key == 76 then -- unmount from Mia
                        model:unmountObject(picked_model,
"")
                  end
            end
      end
end
```

If an object is to be mounted to Mia, the mountObject function is called on the Mia model in VT_Core's SceneManager file. Presently the selected model and a string representing where the object will be mounted are the only two parameters of mountObject. In the future, the offset position and orientation may also be passed in.

Next, the program calls the MountObject function in GrabbingFunctions.h of VT_Mocap's mocap plugin. The function detaches the model from the scene and attaches it to Mia via Ogre functions in C++. The scene node where the model was previously located is stored in a map to be used when the object is unmounted. Finally, the onMount function is called. It sets the variable that the model was mounted and synchronizes the physics body and model position with the entity position. If the programmer has to change anything about the model right after it is mounted, those changes can be made in onMount.

If an object is to be unmounted from Mia, the unmountObject function is called on the Mia model in SceneManager. Currently the selected model and an unused string representing where the object was mounted are the only two parameters to unmountObject.

Next, the UnmountObject function in the mocap plugin is called. By calling Ogre functions, UnmountObject detaches the mounted model from Mia and attaches it back to the scene where it was originally stored. The scene node where the model was previously located is retrieved from the map. UnmountObject uses the scene node to move the object back to its original position. Finally, the onUnmount function is called. It resets the mounted boolean in the model and synchronizes the physics body and model position with the entity position. If the programmer has to change anything right after the model is unmounted, those changes can be made in onUnmount.

The mounting functions in VT_Core's SceneManager.lua and VT_Mocap's GrabbingFunctions.h are shown below:

**SceneManager.lua (VT_Core)**

```
---------------
-- mountObject(): This function is called when one object is mounted to
another.
-- Params:  model - model that is being mounted to this object (self)
--          node - specific mount node (of self) at which to mount
model
---------------
function Model:mountObject(model, node)

      if model ~= self then   -- don't mount to self!
            if model.mounted then
                  self:unmountObject(model, node)
            end
            mocap.MountObject(self.entity.mImpl, node,
model.entity.mImpl)
            model:onMount(self, node)
      end
end

---------------
-- onMount():     Updates the model's internal data, upon being
mounted.
-- Params:  model - model that self is mounted to
--          node - mount node (of model) at which self is mounted
---------------
function Model:onMount(model, node)
      self.mountParent = model
      self.mounted = true

      -- sync data between self and its entity
      self.position.x = self.entity.x
      self.position.y = self.entity.y
```

```
        self.position.z = self.entity.z

        -- set the physics body position
        model.entity.physics.body.position = self.position
end
```

**GrabbingFunctions.h (VT_Mocap)**

```
//-----------------------------------------
// MountObject(): Mounts one object to another.
//
// self - The Ogre entity representing the object being mounted to.
// boneName - The name of the bone on self to mount the object.
// mountObj - The Orge entity representing the object to mount.
//-----------------------------------------
void MountObject(Ogre::Entity* self, const Ogre::String& boneName,
                         Ogre::Entity* mountObj)
{
        // add/update mountObj's parent node in the map
        sceneNodeTable[mountObj] = mountObj->getParentSceneNode();

        // move mountObject from scene node to self's bone
        mountObj->detatchFromParent();
        self->attachObjectToBone(boneName, mountObj);
}
```

## *Luster Issues + Bugs*

Presently, objects are mounted *through* Mia's hands. We tried to change the position offsets of mounted objects so it would appear as if Mia's hands were holding these objects. However, the objects still would not change position even though we passed their offsets into Orge functions via the MountObject. Possible further work would be to try getting these offsets to work.

Another issue that we have presently involves highlighting objects when unmounting. First note that an object is selected only when it is highlighted. Assume the user has selected an object and mounted it. Then assume the user deselects the mounted object. If the user wants to re-select the object, the user must highlight the object in the position *before* it was mounted. This is counterintuitive since the user would expect to select the model by highlighting it in its current position on Mia, not in its previous position before being mounted.

Both of these issues may be related and could have a common fix. We think the issue is related to the fact that the bones of the models are not in the same transform space as the model's root bone. Thus, when an object is mounted to a bone and offsets or orientations are applied to it, they are done with respect the incorrect bone position.

## *Limitations + Further Work*

One limitation mentioned earlier was applying position and orientation offsets. SceneManager's mountObject currently has just two parameters: the object to mount and a string representing where the object will be mounted. In the future, the offset position and orientation may also be passed in. Assuming the Ogre offset problem was fixed, this would allow each grabable object to have its own custom offset.

We would also like to implement functionality to catch errors from Orge function calls inside mocap plugin's MountObject and UnmountObject functions. If an object fails to mount or unmount properly, the Ogre engine would throw an exception. At the moment, nothing is done to handle these exceptions. Ideally, our C++ functions should handle these exceptions gracefully without ending the program prematurely.

Finally, the user should be able to right-click on a grabable model to display a pop-up menu. The menu would show all of Mia's mountable body parts. The user can then choose which body part on the Mia model to mount the object. To unmount objects, a different menu could be displayed allowing the user to click on any mounted objects in order to unmount them. At the moment, only the keyboard is used to mount and unmount objects.

# Luster Mocap Plugin

## *Introduction*

The purpose of this document is to describe how the Luster platform receives motion capture data from Autodesk's MotionBuilder, and applies it to a 3D character model. It is assumed that a functional MotionBuilder plug-in exists. Said plug-in acts as a server with the ability to read motion capture data from a suit, and stream it over the internet. Details on how this plug-in works is outside the scope of this document.

## *Background Information*

Live data is received in Biovision Hierarchy (BVH) format. This data is typically stored as plain text files with a .bvh extension. Their content includes:

1. A header, which contains a hierarchy of all the model's joints
2. The number of frames
3. The amount of time between each frame
4. Successive lines of numbers, where each line represents a frame. The numbers specify translation and rotation values for each of the elements specified in the header.

The basic idea is to receive live data over the Internet in the above format, and use it to animate a 3D model in real-time. The Luster platform uses Ogre3D (www.ogre3d.org) as a means of interfacing to system graphics hardware without requiring high knowledge of Direct3D or OpenGL. Therefore, as live motion data is being received, Ogre3D functions are called to apply rotations to the bones of the current 3D model. This is achieved by writing a Luster plug-in with the ability to invoke low level C++ code. Such code is responsible for live interfacing between the MotionBuilder plug-in and Ogre3D under Luster. Comments within the code are plentiful, which should provide a good understanding of its execution. Note that the abundant #ifdef instructions are there only for portability between Win32 and Mac.

## *Live Data Reception*

An Internet connection with the MotionBuilder plug-in begins by establishing an IP connection over port 12345, and requesting the 3D model's BVH header, followed by a stream of frames. The header information is received over TCP, and successive frame information is received via UDP. This is because the header must be received reliably, whereas occasionally losing a frame is acceptable. The header is received over several packets, since in some cases it may be too huge to fit in a single one. Boost libraries are used to handle TCP (for the header), and Winsock is used to handle UDP (for the motion

data). Lastly, since the same port is used for both TCP and UDP, the TCP port on the client side has to be closed after the header is received.
The header data is packaged into a string array that is parsed to get the joint data. Some joints have 6 channels (including X, Y, Z for the position along with X, Y, Z for rotation). Each joint is then stored in a vector of joints along with their necessary information. This is done so that incoming packet data can be read as channel data, and be applied to the corresponding joint by matching their names. This requires that the model used in Luster be similar (if not the exact same) to the model whose data on the MotionBuilder side. It should be noted that the "Mia" model, loaded in Luster, lacks the huge artificial bone on her back that is present in MotionBuilder. This data is just ignored, as the Luster model does not have a matching name.

## *Intrgration into Luster*

Application of live data into a 3D model requires creation of quaternions for each rotation. Therefore, a quaternion is created for each incoming axis rotation, which are then multiplied together prior to being applied to a bone. A loop is used to iterate through a model's bones, looking for a handle match between the current bone and a joint stored in the header. If a match is found, a quaternion is created and applied to the bone. Please note that bones must be set as manually controlled. Quaternion creation, and application of a live rotation data, is done as such:

```
Quaternion quat_I = Quaternion::IDENTITY;
Quaternion quat_x = Quaternion(live_X_rotation,Vector3::UNIT_X);
Quaternion quat_y = Quaternion(live_Y_rotation,Vector3::UNIT_Y);
Quaternion quat_z = Quaternion(live_Z_rotation,Vector3::UNIT_Z);
Quaternion quat = ( (quat_I * quat_x ) * quat_y) * quat_z;
currentBone->rotate(quat);
```

## *Future Improvements*

A mocap object should eventually be returned to the luster side, so if there were more than one mocap connection they could be managed by the luster scripts. A mocap object may also be used for error checking. The luster mocap plugin should also send a disconnect signal to the MotionBuilder plugin, so the connection may be closed cleanly. This way, the same computer may connect and reconnect multiple times.

# Models + Lighting

## *Summary*

The objective of the model and light loading interfaces is to provide the infrastructure to hold important initialization information such as starting color, position, orientation, scale, material properties and other metadata. The loading functions were made to perform asynchronous loading of meshes and materials transparently to the programmer, as well as to transparently perform complex loading issues, such as loading the same mesh for multiple “models” in a scene.

## *Design*

### Scene Manager

The Scene Manager provides the interface for all of the models in the scene. It is the single instance from which the scene and stage objects internal to luster can be accessed. It contains the definition of a model with all of its metadata, and is called to load the initial models. It provides the highlighting functionality for the various models loaded into the scene, transparently accessing metadata for every model to determine if a particular object is selectable by the puppeteer to play animations on, or grabbable by the actor. Finally, the Scene Manager handles the physics world used by the virtual theater, needed to support the raycasting for highlighting objects.

### Light Manager

Similar to the Scene Manager, the Light Manager contains the interface for accessing all the lights in the scene. It also defines the wrapper for the lights, with all the necessary metadata regarding a light’s position, color, orientation, type, attenuation, etc.

### Mesh Manager + Material Manager

These managers are responsible for the asynchronous loading of the meshes and materials required by the models, with error checking to prevent the same mesh from being loaded twice.

## *Functionality*

All models and lights specified by the XML are loaded into the scene at their correct positions, with the correct color, orientation, scale, and other various properties. Once the scene is loaded, the Scene Manager handles the highlighting and selection of ‘selectable’ and ‘grabbable’ objects.

## *Usage*

```
getScene()
     - Returns the luster scene
```

```
getWorld()
     - Returns the luster physics world
getPicked()
     - Returns the highlighted object
setMocapModel(mocapModel)
     -  Sets which model is the mocap model
     -  The mocap model to be set
getMocapModel()
     - Returns the mocap model
mouseOver(camera, clientType)
     -  Callback from within update() for highlighting on
        mouse over
     -  Takes the camera to perform raycasting, and the
        client type to determine if client (actor, audience,
        puppeteer or stage manager) is allowed to highlight
        models in the scene
```

### *Luster Issues + Limitations*

Models with sub-entities have the meshes for those sub-entities applied correctly, however if the model is highlighted by a puppeteer or actor client, the sub-entities of the model will not be assigned the correct material. This is a bug, not a luster limitation, and can be fixed by adding code to the highlighting functions to re-apply the necessary materials.

### *Further work*

Currently only one mocap actor is supported. It is possible to have multiple mocap actors, but will require a rewrite of some of the luster-side mocap framework.

# MotionBuilder Plugin

## *RIT VT Server*

*RIT VT Server* is a plug-in for MotionBuilder that gets motion data from a character and streams it to clients over the network. Clients can apply the data back to the model to view the motion in real-time.

## *Design*

RIT VT Server consists primarily of one main class called RITVT_Server. It uses one main thread, as well as two helper threads that handle incoming requests. Each thread may use one or more of the utility classes *SocketTCP*, *SocketUDP*, and *Broadcaster*.

The main thread's job is to send motion data to the clients. It does so with the *DeviceIONotify()* method, which MotionBuilder calls at an interval corresponding to thirty frames per second. The format of the data is the translation and rotation for the root bone and the rotation for all other bones. Both values are given as three decimals in X, Y, Z order. An instance of the *Broadcaster* class actually handles sending the data to all clients.

Meanwhile the two helper threads listen for requests, the first with *SocketTCP* and the second with *Broadcaster*. When the first thread receives *h* it sends the BVH header for the character. Any clients sending a request matching *m* to the second thread are added to the list in *Broadcaster*. On the next frame they will start receiving frames from the main thread.

## *Dependencies*

Before attempting to build the project, make sure you have the Boost libraries installed. They are available from boost.org or boostpro.com, the latter of which has an installer for Windows that can download and install the required precompiled libraries. Make sure to select the libraries for *Boost Date/Time* and *Thread* for your version of Visual Studio.

Of course, MotionBuilder's *OpenRealitySDK* is also needed. The plug-in references the *fbsdk.h* file in *[MotionBuilder]/OpenRealitySDK/include* and links against *fbsdk.lib* in *[MotionBuilder]/OpenRealitySDK/lib*. Note that *fbsdk.lib* is sometimes not included in the academic version of MotionBuilder and may need to be obtained directly from Autodesk.

## *Building*

The first method for building the project is to use the Visual Studio project file. Simply open the project file and issue the *Build* command. Note you may need to change the project settings based on where MotionBuilder and/or Boost is installed, and the output directory will need to be changed to the *MotionBuilder\bin\plugins* (your exact

directory path may differ) directory so that the *ritvt_server.dll* file is automatically generated in the correct location.

Alternatively, the *makefile* can be used if *GNU Make* is available. On Windows, it's recommended to install *Cygwin* and use the *Visual Studio Command Prompt*.

## Running

After successfully building the project, *ritvt_server.dll* will be located in the *MotionBuilder\bin\plugins* directory. Open MotionBuilder and load a scene with a character in it. Next, in the *Asset Browser* tree, click *Devices*. Scroll over to *RIT VT Server* and drag it to the *Navigation* window. Click the *Online* button. RIT VT Server is now waiting for requests.

Furthermore, a sample client is included. Open a command prompt and enter *client host port*, where *host* is the address of the computer running MotionBuilder and *port* is 12345.

## Issues

Boost was used because of cross-platform nature. Unfortunately Boost does not provide a decisive way to kill a thread for precisely that reason, and a mix of asynchronous reads and interrupt calls were needed to end threads. The code could be simplified if this capability is ever added.

Other improvements that could be made include allowing for multiple characters and using multicast to send out to multiple clients. Currently *Broadcaster* simply loops through a list to do so.

# Networking

## *Summary*

The networking subsystem is broken into three parts: the core, the client, and the server. The core functionality of the system consists of the various message classes (VT_Core/Src/network). This, and the rest of the Virtual Theater core, is all the networking code that is shared between client and server. Message handling and other logic is divided between the two distributions (VT_Client/Src/network, VT_Server/Src/network), as to allow for independent distribution of clients while maintaining source privacy of the server.
The networking system has been designed with **private-key** encryption in mind. We envision a user ordering a ticket online using a web portal; this ticket would contain a **ticket number** and **authentication phrase**. The client would then authenticate with the server using the ticket number, encrypting all messages using the private key. The authentication scheme has also been set up to handle shows that do not require an authentication phrase at all.

## *Design*

The Virtual Theater networking subsystem has been designed with modularity as its primary goal. We have chosen a proxy like design to process the various messages of the clients and servers. Luster differentiates message types by reading the first byte in the packet; custom messages must begin with an unreserved byte. Each proxy has been designated an unreserved byte to lead its messages. The server/client, upon receiving a packet, inspects this first byte, and forwards the message to the appropriate proxy.

### Server

The server opens a port for each proxy it supports. When a packet comes in, it strips the first byte off and forwards the message to the appropriate proxy.

### Client

The client also maintains an internal structure of proxies. When a packet comes in, it strips the first byte off and forwards the message to the appropriate proxy. The client, unlike the server, also associates an IP address with a proxy. This way, if a default luster packet comes in (CONNECTION LOST, etc), it can be forwarded to the correct proxy using the sender field; non-custom packets can also be handled by the proxies.

## *Functionality*

### Message.lua

Each proxy also has an associated subclass of message. Each message can be built from a constructor or a byte array containing the data. Each message can also

serialize into a byte array to be transferred across the network. This process takes care of writing message type bytes, etc.

## Client.lua

The client class is the only class the rest of the code base sees. This class sets up the connections to the outside world and contains the most abstract methods of network communication. The proxies utilize this class for send, receive, connect, and disconnect functionality. The rest of the code base uses it in a more general way- with methods to authenticate and execute an event.

## Server.lua

Like the client class, the Server class contains the high level network functions (send, received, connect, disconnect). While the proxies perform the message handling logic for this class, the server acts more as an interface to the show; it also contains hooks into the "database" of users for ticket and key validation, keeps track of user authentication information for the other proxies, and acts as a liaison between them.

## Authentication Proxy

The authentication proxy uses a multi-message authentication scheme. The messages must be received in order, or authentication is denied and must be restarted. First, the client knocks on the server. It then identifies itself via ticket number. If the show requires an **authentication key** for the ticket, the server requests the key from the client. The client replies by utilizing its **authentication key** to encrypt all further communication. The client identifies its user type to the authentication server. At any point the server authentication process can be denied. A byte code reason is sent. On acceptance, a unique **key** is sent to the user. The user must use this key and say Hello to a proxy before using it.

## Event Proxy

When authentication is complete, a client may begin using its event proxy. This proxy takes events and forwards them to the server. The server takes these events, makes sure the user has appropriate permissions, and then forwards them to the other clients. Event messages can either be string id's of Cues, or serialized CueEvent objects. The server event proxy only accepts messages from authenticated users. A user registers itself with the event proxy (and any other proxy) by sending a hello message. It sends a unique **key** to the proxy telling it to associate it with that ip. The server then sends back the address in which to communicate on for that proxy.

## *Usage*

From the non-network team perspective, the client class is the interface to all communication. It is implemented as an EventDispatcher. Objects are to listen to the various events this class can dispatch-

Client.AUTHENTICATE_SUCCEED
Client.AUTHENTICATE_FAILED
Client.CONNECT_AUTH
Client.DISCONNECT_AUTH
Client.EVENT_READY
Client.CONNECT_EVENT
Client.DISCONNECT_EVENT

Generally, only AUTHENTICATE_SUCCEED, AUTHENTICATE_FAILED, and EVENT_READY are needed. Some useful functions to call on client are Client:executeEvent( id ), and Client:authenticate().

## *Luster Issues + Limitations*

Luster's main limitation is that only 1 peer object can be created per runtime. This does not seem to be a Raknet issue, but a Luster wrapper issue. Because of this, the single peer instance acts as one big interface to all the open sockets. It is impossible to tell which socket a packet came in on. Also, a client cannot connect to multiple sockets on the same peer- an ALREADY_CONNECTED message is sent back. This implies that once a user is authenticated, it must disconnect from the auth server before connecting to the event server (if running on the same machine). This also restricts us to only one running proxy per runtime. Had we known this upfront, we would have had all the proxies run on the same port- this will be discussed in further work. This also seriously cripples our ability to multicast. We can only multicast to every client connected to the peer- we cannot only multicast out a single port.
The core library is loaded at runtime, as opposed to load time. This restricts the use of core constants in the global (or file) name space. We have gotten around this by hand including the necessary files in the dependency tree.

## *Further work*

### Show Database

A database and hooks into it should be developed for storing and retrieving show information such as ticket information, client type information, and key authentication information. Ideally, a user would purchase a ticket online, or an admin would set up staging/puppet accounts which would all be stored in a database. The server should query this database for all show information.

### Encryption

Private key encryption has been designed into Virtual Theater, but not implemented. We need a way for the Client to dispatch an event that prompts the user when a key is requested from the server. The encryption logic must also be added to the message class. The key look-up logic is already in the server. Another scheme which is

supported by Raknet, but not Luster, is a public-key scheme. Raknet has some sort of built in secure communication channel.

## Client/Server Synchronization

To save on bandwidth, we have implemented a Cue and Event system. Rather than sending the orientation and position of every object on the stage (as they move), we just pass around predefined events for the clients to execute. A tween turns into the sending of a string, rather than constantly updating orientations. Our approach has had a drawback, though, of not keeping a persistent state among the clients and server. If a client connects late to the show, it will be out of sync with the server and other clients (as it has not executed the previously executed cues). The mocap wearer's position and any grabbed items will also be out of sync. To work around this, we have added a locking mechanism for the show; once the show begins no more clients may connect. The client and server should synchronize when a client is authenticated. A limitation would be sounds- since a sound cannot begin playback in the middle of a file.

## Distributed Server Environment

The reason for such a complicated authentication scheme was to support distributed servers. We planned a proxy to be made to handle inter-server communication and synchronization. In the future, we expect clients to connect to the mocap or event servers closest to them, and for these servers to load balance themselves. This is why the authentication server sends back the address of the proxy being requested, rather than just a port. If each proxy is run on a different server (or in a different process), this could solve many of the aforementioned limitations. A solid inter-server communication peer-to-peer proxy would have to be developed.

## Channels

Messages can be guaranteed to be received in order through the use of channels. Each proxy should probably be given its own channel as not to interfere with each other.

# Object Highlighting

## *Summary*

The first part of the picking function is to highlight selectable objects. The second part of the picking function is to determine which object has been picked. When the user mouses over an object it will be highlighted by changing its color.

## *Functionality + Usage*

The picking function takes the camera as a parameter. It then uses this and the physics raycasting functionality of luster to send a ray into the scene from the cursor to determine what object, if any, the mouse is over. The raycasting function returns a listing of all the objects the mouse is over. Those objects are physics bodies and that must be accounted for in the rest of the code.

```
local results = luster.physics.raycast({camera = camera, world =
self.world})
      if results and #results > 0 then
            local body = results[1].body
            local model = self:getModelByEntity(body.entity)
```

With this code, the array of objects that the mouse is over is stored in results. Since the objects in the array are ordered based on which is closest to the camera, and we want the closest object, we store the first object in a local variable. Since body is just a physics body, we call a function that gets the model of the object from its physics body. The rest of the code does checks to see if model is set correctly and if the object currently under the mouse is selectable. The function also stores a previous node for determining if the mouse moves between two objects without hovering over no objects first.

## *Luster Issues + Bugs*

Since this function relies on the physics raycasting function, if the physics system is changed through a runtime update, the picking function may need to be updated to reflect those changes.

## *Limitations + Further Work*

As of now, the highlighting function relies on changing the material of the selected object to a highlighted color. Being able to dynamically change the color properties of the current material would be an improvement.