An Introduction to X-VP

Table of Contents

Table of Figures

1 Overview

The Extensible Visualization Platform (X-VP) consists of a unique set of objects and API's that allow a developer to create 3D graphics applications such as games and other types of visual simulations with audio, input and network capabilities. X-VP's design promotes flexibility and extensibility while still providing a significant amount of functionality right out of the box. This document serves as an introduction to X-VP and devotes several sections to outlining the major components of the engine.

X-VP is fundamentally different than other types of graphical SDKs. It doesn't corner the developer into using restrictive, pre-defined functions. Instead, it focuses on providing foundations upon which the developer can build their own unique application. That's not to say that X-VP doesn't contain high-level functionality that can assist the developer in rapid application development. The difference is that while X-VP provides functionality in many high-level areas, it also provides a customizable framework that can be used to extend, or even completely bypass the standard engine functionality.

X-VP has been architected from the ground up with the following design principles:

- **Functionality** - X-VP provides the developer with an incredible amount of functionality so they can hit the ground running. The engine ships with everything from basic polygon manipulation and math libraries to Skin Mesh/Skeletal Animation and Particle Systems.
- **Flexibility** - Through the use of X-VP's unique Node/Controller System, not only can the developer leverage existing features of the engine, but can also extend engine functionality and even create brand new objects and behaviors using X-VP's object framework. This allows the developer to control the engine in countless ways to suit their specific needs.
- **Portability** - X-VP uses an abstraction layer to shield the developer from the platform and environment, but at the same time, provides the option to control these low level components if so desired. This provides for a simple and smooth porting process while promoting the maximum amount of code reuse. Code that is written for X-VP can run on any platform that has an "X-VP Support Package" (XSP) available.
- **Compatibility** - X-VP ships with integrated tools and exporters for the most popular 3D content creation packages such as 3D Studio Max. These tools streamline the asset exportation process with incredible ease.

2 Features at a Glance

Graphics / Rendering

- Integrated Vertex/Pixel shader pipeline and fixed-function pipeline
  - Reference Toon Shader
  - Reference Cubic Environment Mapping Shader
  - Reference Phong Lighting Shader
  - Reference Per-Pixel Lighting Shader
  - and more...

  XVP shaders can be written within the Node/Controller framework allowing them to be attached to and detached from nodes in the 3D Scene Graph. For example, the Reference Toon Shader shipped with XVP can be attached to a model of a 3D tank (or individual pieces of the tank). From that point on, affected pieces will be rendered according to the Toon Shader. Simply detach the Toon Shader from the tank to cause the toon shading effect to stop and rendering to default back to the fixed-function pipeline.

- 3D Scene Graph for building complex scenes with parent-child spatial relationships
- 3D Hierarchical Models
- Skin Meshes/Deformable Models via skeletal deformation
  - Hardware skinning/hardware lighting
  - Software skinning fallback
- Particle System and Extensible Framework
- Camera Systems
- Per-object Render State Management with intelligent batching algorithm for more efficient render state changes.
- 2D Sprites with animation system
- 2D Scene Graph with Z order management for sprites, movies, etc..
- Movie Playback - AVI, MPG, etc...
  - Texture and screen based
- Animated Cursors
- Font system
- Extensible User Interface Framework supporting 100% graphically customizable user interface objects such as:
  - Sliders
  - Push Buttons
  - Toggle Switches
  - Check Boxes
  - Progress Bars
  - Movies
  - Animated Sprites
  - Text Boxes with formatting capabilities such as left, right and center justification and automatic word wrap, etc...

  The framework automatically detects resolutions and resizes controls appropriately. This means that you define the screen layout once, and the framework automatically handles all other resolutions. It also recognizes events such as mouse up, mouse down, mouse over, etc. Developers can create custom controls as well as custom events.

Animation

- Animation System supporting rotation, position and scale key frames.
  - Animation can be applied to any object in the engine including models, skin meshes, cameras, particle systems, custom objects etc. Animation tracks can also be shared between multiple objects.
  - Event system that allows creation of animation events/triggers.
- Multiple Animation Track Blending for seamless transitions and mixed animations
- Physics Controller
- Look-At Controller
- View Bind Controller
- First Person Controller
- Texture Animation Controller

Audio

- 2D Sound playback
  - Volume, Pitch, etc...
- 3D Sound playback
  - Position, Velocity, Rolloff, Doppler, etc..
- Streaming Music playback
- Priority-based sound system
- Optional FMOD reference driver available. (Requires seperate FMOD license)
  - Music playback including:
  - MP2/MP3/WAV/WMA/ASF/Ogg Vorbis/AIFF/MIDI Support.
  - D3v1, ID3v2, ogg vorbis, ASF tag support.
  - Mod Playback. MOD/S3M/XM and IT. MOD music synchronization.
  - CD Playback and Internet Streaming Support.

Networking

- Network support including custom multi-threaded networking API
- Highly configurable client/server network model built on top of X-VP's networking API supporting:
  - Automatic packet pooling and memory management / memory fragmentation protection
  - Automatic game client/lobby management
  - Encrypted communication via RSA Encryption
  - Password protected game sessions

Input Devices

- Input Device support including:
  - Joysticks
  - Gamepads

    - Keyboards
    - Mice

Miscellaneous Functionality

- Math Libraries including objects such as 2D,3D and 4D vectors, 2x2,3x3,4x4 matrices, quaternions, planes, rays, etc...
- Geometry Intersection Libraries
- Collision Libraries
- Built-in application state manager
- Integrated scripting system via Lua 5.0.2 (www.lua.org)
- Timers
- Action Maps
- Color Conversion Object
- Smoothing Filter
- RSA Encryption/Decryption
- Resource/Media packaging
- Logging Functionality
- Support for underlying render API. I.e. DirectX, OpenGL, etc...

Tools / Exporters

- 3D Studio Max Exporters for several objects including:
    - Hierarchical Models
    - Skin Meshes
    - Animation Tracks
    - Lights
    - Cameras
    - Particle Systems

3 Core Components

This section describes the core components of X-VP. The core components include Scene Nodes, Scene Node Controllers, Scene Graph, Math Library, and Exporters.

3.1 Scene Nodes

Scene Nodes are the core engine objects. They are referred to as "nodes" because they're assembled into a node-based scene graph within X-VP. Nodes can be anything from a camera to a 3D model to a skin mesh to an Octree data structure. X-VP comes with many types of pre-built nodes and allows the developer to create new nodes that simply "plug" into the existing architecture with minimal effort. This is an extremely powerful concept when coupled with controllers. See below.

3.2 Scene Node Controllers

Think of these objects as "behaviors". They're called controllers because they "control" the nodes they're attached to. Controllers can control the movement and render properties of a node. A simple example of this relationship follows. You could attach a "wind" controller to 3D objects in a scene. When the 3D objects are updated, the “wind” controller influences them. The implementation of the "wind" controller could simply add a wind vector to the position of all nodes that it's attached to. Any number of controllers can be attached to any number of nodes; i.e., nodes and controllers have a many-to-many relationship. X-VP comes with many types of controllers. Also, the developer can create new controllers that attach to any type of node, creating new functionality in the existing engine framework.

3.3 Scene Graph

Internally, X-VP manages a Scene Graph. The Scene Graph is the hub of the engine. Controllers, nodes and other objects congregate inside the Scene Graph to be managed by the engine automatically. The Scene Graph maintains parent/child relationships between nodes that exist in the engine.

3.4 Math Library

X-VP comes with an extensive, object-oriented math library that is completely platform independent. Among other functionality, it supports matrices, quaternions, vectors, polygonal operations and intersections, etc.

3.5 Exporters / Tools

A major focus during X-VP's design was on export tools for the most popular 3D content creation packages. X-VP comes with export tools for 3D Studio Max. With the tools, you can easily export animation sequences, models, skin meshes, cameras, lights, particle systems, etc.

The figure below illustrates the relationship between core components and the rest of the system.

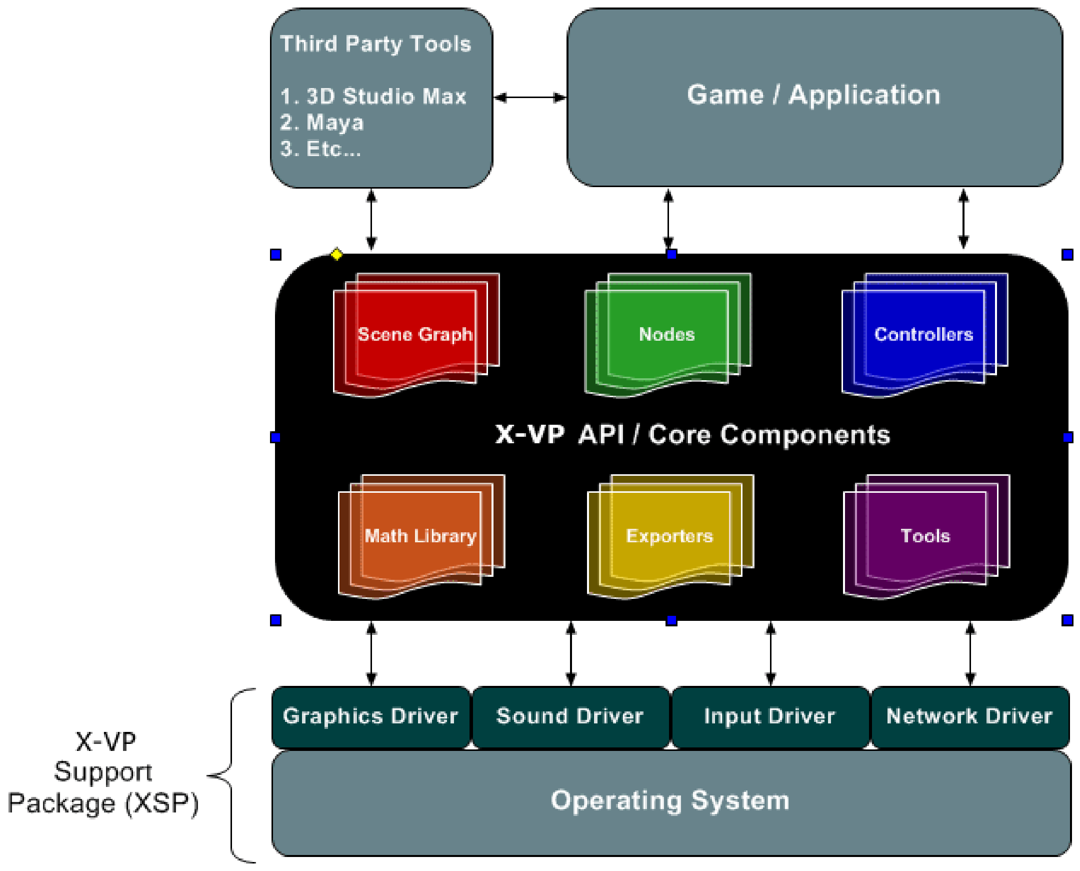

Figure 1 - X-VP Core Components

4 X-VP Support Package (XSP)

An X-VP Support Package (XSP) is a hardware and API abstraction layer that allows X-VP to run on a wide array of platforms such as Windows, Mac, Linux, etc. Once an XSP is written for a particular platform, the upper layers of X-VP simply "plug" into the XSP.

It accomplishes this abstraction via driver objects that abstract environment specifics away from the high level logic in the engine. The drivers include:

- **Graphics Driver**
- **Audio Driver**
- **Input Driver**
- **Network Driver**
- **OS Driver**

The driver objects and their associated APIs are directly available to the developer and form the foundation of the application. The list below describes a few of the operations supported by each of the drivers. The list of driver operations is far from complete, but should give you an understanding of the purpose of each driver.

- **Graphics Driver**
  - Graphics Device Configuration
  - Device Render State Management
  - Vertex Buffer Management
  - Index Buffer Management
  - Texture Management
  - Movie Management
  - Light Management
  - Vertex/Pixel Shader Management
  - Etc…

- **Audio Driver**
  - Audio Device Configuration
  - Sound File Management including 2D/3D and streaming sound
  - Etc…

- **Input Driver**
  - Input Device Configuration for Keyboards, Mice, Joysticks, etc.
  - Input Device State Polling
  - Etc…

- **Network Driver**
  - Network Connection Management via UDP and TCP
  - Messaging Mechanisms
  - Etc…

- OS Driver
    - Typical Operating System Functionality
    - Time keeping methods
    - File/Directory Access
    - Etc…

The next few sections describe each of the drivers in more detail. For a complete list of methods supported by each driver, see the X-VP API Reference.

4.1 Graphics Driver

The Graphics Driver provides an interface for objects that need access to the display device. Through this interface, the driver exposes services such as resource management, device settings and other graphics related services.

Resource Management
The graphics driver manages resources such as textures, vertex buffers, index buffers, and vertex/pixel shaders by providing methods to create, use and destroy these objects. By calling the methods of the Graphics Driver, higher-layer objects such as models and meshes, can allocate and use textures, vertex buffers and so on without any specific knowledge about the underlying software/hardware platform.

For example, say that you have a 3D model of an F-16 and that it requires three different textures to render correctly. Typically, the application will request the Graphics Driver to create the three textures. After the Graphics Driver creates the textures, it passes handles to each of the textures back to the model. The model can then use these handles to set the textures for rendering the F-16, or it could use the handles for any other supported texture functions within the Graphics Driver.

This is the general process involved when the application uses the Graphics Driver. Higher-layer objects such as models, skin meshes, particle systems, or even custom user objects can create their own resources through the Graphics Driver.

The following process outlines the general procedure used to create resources with the Graphics Driver.

1. Make a call to the Graphics Driver and request allocation of a specific type of resource such as a texture, vertex buffer, index buffer, render state, etc…
2. With the handle received from the allocation call, use the available resource methods during runtime to control the resource. For example, with the handle to a vertex buffer, you can lock, fill and unlock the buffer.
3. When the resource is no longer needed, request deallocation of the object from the Graphics Driver.

Note: The Graphics Driver will automatically clean up any remaining resources when the engine shuts down, even if the object that originally asked for resource allocation forgets to tell the Graphics Driver to destroy the resource. This functionality is in place to prevent memory leaks.

The Graphics Driver intelligently handles multiple allocations of the same resource. For example, if an allocation request is received for a texture that has already been created, the driver will not load two copies of the texture into memory, but will instead manage the single copy as if it were multiple copies. This allows multiple objects to use the same texture without duplicating system resources.

Render States
The Graphics Driver is responsible for managing the render state of the rendering device. In other words, the driver controls render states such as:

- Lighting Parameters
- Fog Parameters
- Alpha Blending Parameters
- Texture Stage Parameters
- World, View, Projection Transforms
- Etc…

Each render state is exposed through methods defined in the driver object. Many of the higher-layer objects use these methods to set up their specific rendering states before making a rendering call. For a complete list of supported render states and their associated methods, see the X-VP API Reference.

4.2 Audio Driver

The Audio Driver is responsible for the configuration and management of the audio device in a particular hardware environment. It also provides an interface to the application from which audio can be created, played and destroyed.

Resource Management
The driver manages traditional 2D (stereo) audio, 3D audio, as well as streaming audio.

This driver functions almost identically to the Graphics Driver. That is, when the Audio Driver receives an allocation request for a sound, it creates the sound and passes back a handle to the sound object. The caller can then use this handle at runtime to play, stop or otherwise manipulate the sound object. For a complete list of sound related methods, see the X-VP API Reference.

Also, like the Graphics Driver, if multiple requests are received for the same sound files, only one sound file is loaded into memory. This does not apply to streaming files.

4.3 Input Driver

The Input Driver exposes input devices such as keyboards, mice, joysticks and game pads to the application. It's responsible for enumerating, configuring and polling devices.

Using the methods of the Input Driver, the application can obtain information such as:

- Depressed or raised keys on the keyboard
- Depressed mouse buttons and mouse movements
- Depressed joystick/game pad buttons and axis movements

See the X-VP API Reference to learn more.

4.4 Network Driver

The Network Driver manages network connections for the application. For example, the application can utilize the driver to create or listen for TCP connections as well as create UDP connections. The driver also contains methods for sending and receiving data over connections.

X-VP comes with a Client/Server network model that is built upon services of the Network Driver. In addition to the base network services provided by the driver, other objects are included that can be used to manage memory pools for efficient network packet allocation and deallocation as well as byte stream readers and writers that effectively send logical packets over stream oriented TCP connections.

See the X-VP API Reference to learn more.

5 The Scene Graph

At the heart of X-VP is the Scene Graph. It's a hierarchical collection of scene nodes stored in a tree data structure. The tree data structure makes it possible for parent nodes to pass information along to their children in a consistent manner.

5.1 Scene Nodes

First, let's define exactly what a scene node is. A scene node is an object that has properties such as a world transform in 3D space, vertex buffer, index buffer, render state, and so on. Scene nodes are usually the base object of a more specific object such as a 3D model, bone, camera, particle system, or skin mesh, etc. Since the Scene Graph is a collection of scene nodes, by definition, it is a collection of any object that is derived from a scene node. You can create new scene node types by deriving from the base scene node object and providing custom implementations.

Now that you understand that a Scene Graph is a collection of scene nodes, let's define a few rules and operations that apply when adding a scene node to the Scene Graph. First of all, the Scene Graph is updated and rendered every frame. This means that each scene node in the Scene Graph will be sent a message to update and render itself. In order to do this, the Scene Graph must be traversed and each scene node must be visited by the traversal. Since the Scene Graph is organized into a tree structure, a depth-first traversal is used to visit each scene node.

So, what does a scene node do when it receives messages to update and render itself? Well, the answer is…whatever its implementation specifies. By default, a scene node will do nothing in response to an update or render message. The update and render functionality is specific to the type of scene node being visited and is typically added when deriving from the scene node base object by overriding methods to handle these messages.

Earlier it was noted that the Scene Graph structure allows parent nodes to pass information along to their children. This information consists of position and orientation in 3D space. As a result of this communication, children will inherit the motion of their parents. This allows you to build complex hierarchies of objects that inherit motion in various ways.

By default, the Scene Graph will always have at least one scene node. This is because, during initialization, the engine creates the first scene node as a "dummy" node that will always be the root of the Scene Graph. In fact, you can never remove this root node. Therefore, when you add scene nodes to the Scene Graph, they will always be children of this default root node.

When you add a new scene node to the Scene Graph, you must specify a parent node for the new scene node. Since the dummy root node always exists, you can add new scene nodes starting at the root.

Another way to think of this default root node is to imagine it as the "world" root node. That is, this node has no transformation, rotation or scaling. In fact, the node's transform matrix is the identity matrix. So, this node will not contribute any positional or orientation information to the rest of the Scene Graph.

5.2 Basic Scene Graph Example

Below is a sample layout of the Scene Graph for a simple scene. This scene is comprised of four scene nodes. For now, we will use generic scene nodes, but later we will investigate a more complex scene where the scene nodes will be more specifically defined.

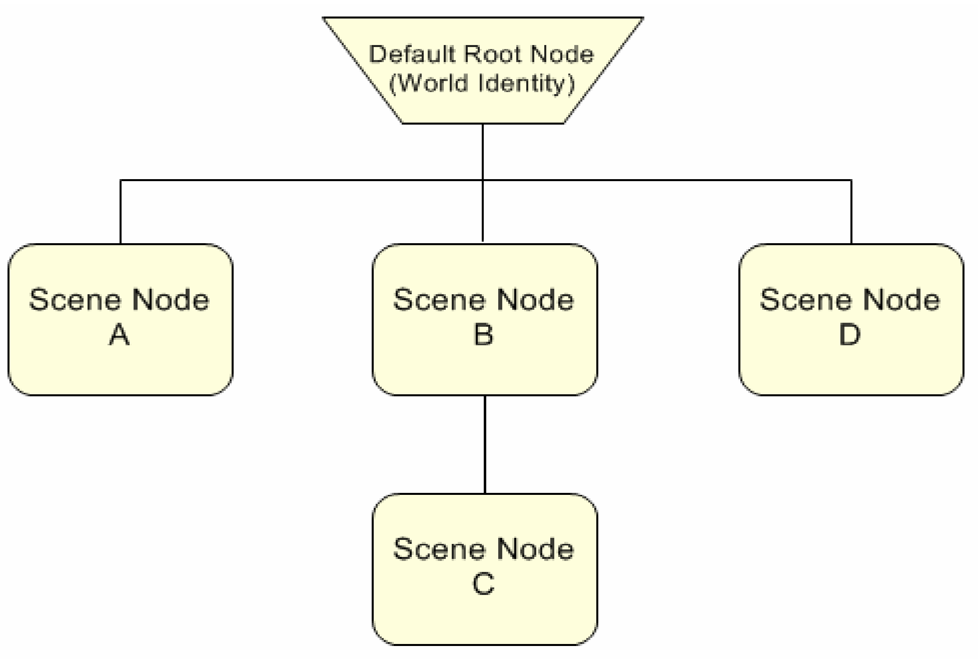

Figure 2 - Basic Scene Graph Example

In this scene, notice how scene nodes A, B and D are attached to the root node. This implies that the A, B and D are attached to the world, but do not inherit any 3D positional or orientation information. On the other hand, scene node C is attached to scene node B. This means that C will inherit B's motion in addition to its own motion.

5.3 Advanced Scene Graph Example

Let's take a look at a more interesting scene graph example. See below.

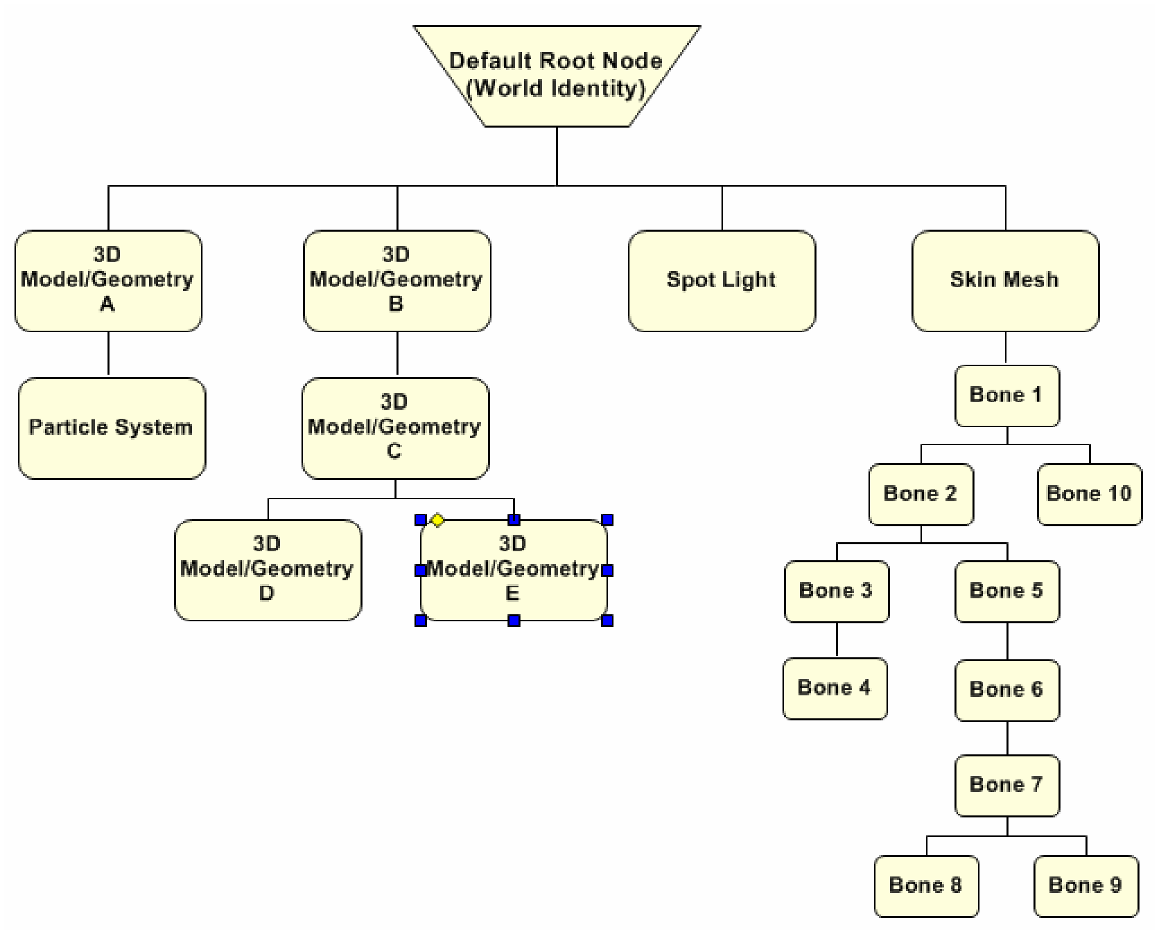

Figure 3 - Advanced Scene Graph Example

Notice in this example, that many different types of scene nodes are used to build this particular scene. There are nodes that represent 3D geometry, particle systems, skin meshes and bones, and even a spot light. For now, we will ignore the specifics of the implementations for each node and instead examine their relationship with one another.

Directly attached to the root node (world) are two 3D Models (A and B), a spot light and a skin mesh. Since these objects are attached to the root node they do not inherit any motion from the Scene Graph. There is a particle system attached as a child to 3D Model A and because of this, the particle system will inherit the motion of 3D Model A. That is to say that the particle system will "follow" 3D Model A in addition to its own motion.

Another important note is that because the particle system is attached to 3D Model A, the world transform of the particle system is effectively relative to the world transform of the 3D Model. This rule holds true for any parent and child relationship. A child's world position and orientation are relative to its parent. Specifically, if the world position of 3D Model A is set equal to (x = 10,y = 0,z = 0), and the particle system's world position is set equal to (x = 10,y = 0,z = 0), the particle system's effective world position is equal to (x = 20,y = 0,z = 0).

Now, for a more complicated object, take a look at the Skin Mesh attached to the root node. The same rules for 3D Model A and the particle system also apply to the Skin Mesh and its children. Each bone attached under the Skin Mesh node will inherit all motion from its parents including the Skin Mesh node itself and any parent bones.

As you can see from this example, the Scene Graph allows you to build complex motion relationships between scene nodes.

5.4 Scene Node Controllers

The previous sections illustrated what a scene node is and how it's used in conjunction with the Scene Graph. For example, you know that child nodes inherit the motion of their parent nodes and that the world transform of a child is effectively relative to the world transform of its parent. However, we haven't explained how scene nodes actually get moved around in 3D space, rendered, etc.

That's where Controllers come in. Controllers are objects that force scene nodes to behave certain ways. For example, a controller could control the position and orientation of a scene node. Likewise, a controller could control the render state of a scene node.

In truth, a controller can modify a scene node any way that it wants. To associate a controller with a scene node, you "attach" the controller to the scene node based on the controller's type. Controllers are categorized into two broad types. They are:

- **Update Controllers** – During the update pass of the Scene Graph, these controllers are notified to modify their associated scene nodes.
- **Render Controllers** – During the render pass of the Scene Graph, these controllers are notified to modify their associated scene nodes.

So, what exactly can a controller modify? Here are a few examples.

1. Position and orientation
2. Render State parameters such as the alpha blending state, textures, lighting parameters, materials, etc…
3. Vertex and Index buffers
4. Vertex/Pixel shader constants

In fact, X-VP ships with physics controllers, animation controllers, texture animation controllers, view binding controllers, various vertex/pixel shader controllers and more. These standard controllers provide basic functionality such as moving scene nodes in 3D space, animating scene nodes with key frame based animation tracks, animating textures, binding scene nodes to the currently active camera, rendering via vertex/pixel shaders, etc.

However, the real power and flexibility of the node/controller design, is that you can develop new controllers that implement new functionality and attach them to any type of scene node.

See the X-VP API Reference to learn more.

6 Application State Management

X-VP interfaces with the Application through application-defined states. Application states are created by deriving from the core state object in the SDK and overriding various methods to respond to events and provide application functionality. States are registered with X-VP through one or more hook points in X-VP's pipeline and are notified when various events occur. Some events each state can respond to are:

- **State Activation** – this occurs when the state has been added to X-VP's pipeline
- **State Deactivation** – this occurs when the state has been removed from X-VP's pipeline
- **State Update** – this occurs once per time step and allows the state to update any time-dependent data. States are only updated while they are registered with X-VP.

All X-VP applications must have at least one state; however, typically there are several depending on the type of application. For example, a video game may have a state for the intro movie, another for the main menu screen, another for normal game play, etc. It's up to the application designer to decide how many states are needed and where they should be registered within the pipeline.

6.1 State Hooks

To understand states more in depth, the figure below depicts the three hooks in X-VP's pipeline where application states can be registered. Each hook occurs at a particular time in the pipeline where an application might want to exert control over X-VP or vice-versa.

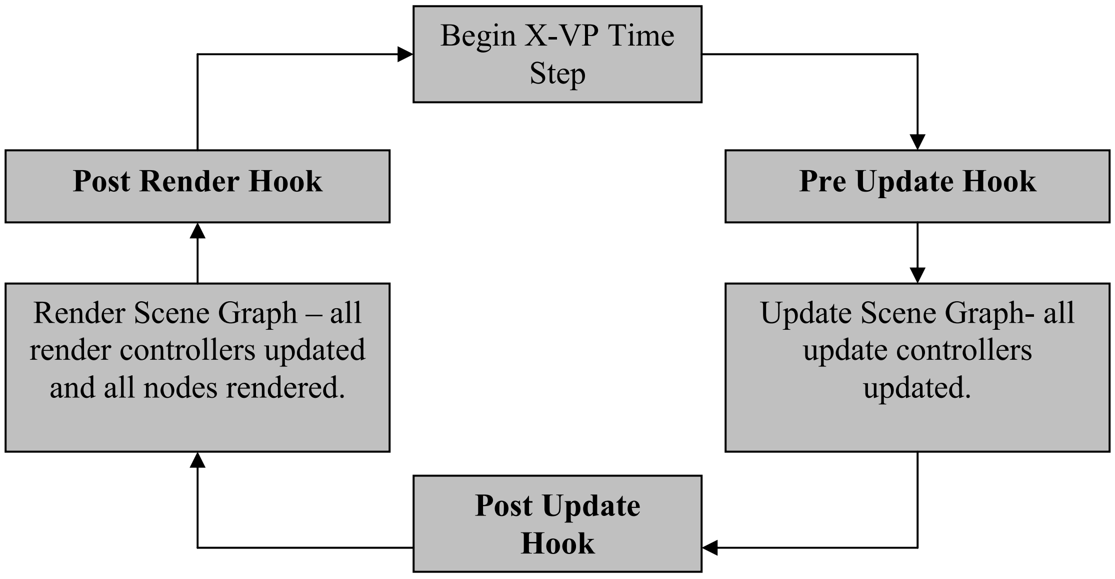
Begin X-VP Time Step
Pre Update Hook
Update Scene Graph- all update controllers updated.
Post Update Hook
Render Scene Graph – all render controllers updated and all nodes rendered.
Post Render Hook

Figure 4 - State Hooks

6.1.1 Pre Update State Hook

States registered at this point in the pipeline will be updated immediately before X-VP updates the Scene Graph. This also means that this hook gets called before all update and render controllers are updated. This is the earliest point in the pipeline where a state can be registered.

6.1.2 Post Update State Hook

States registered at this point in the pipeline will be updated immediately following a Scene Graph update. At this point, all update controllers have been updated. However, nothing in the scene has been rendered.

6.1.3 Post Render State Hook

States registered at this point in the pipeline will be updated immediately following a Scene Graph Render. At this point everything for a particular time step has been completed. That is, the Scene Graph, Update and Render Controllers and Scene Nodes have all been updated to the current time step and all rendering is complete.

6.2 State Management Example

Let's look at a simple example for a typical video game consisting of the following game states.

- **Intro Movie** – contains logic to play the intro movie and check for any user input. If any input is detected, the movie is stopped and the state transitions to the Main Menu Screen.
- **Main Menu Screen** – contains logic for a main menu user interface allowing the player to start a new game, configure options, etc.
- **Load Game Screen –** contains logic that allows a previously saved game to be played.
- **Options Screen** – contains logic that controls a user interface and allows the player to configure game options.
- **Game Play** – contains all game logic and game play elements.
- **Credits** – contains logic to scroll credits.

For the six game states above we could create six X-VP state objects. That is, we could derive from the core X-VP state object six times, once for the Intro Movie State, once for the Main Menu Screen and so on. This is a fairly simple example so we can probably get away with only these six states. However, a more complex example might contain states for other game tasks like loading in art assets from disk (a level loading state) or searching for servers hosting a network game.

Next, we would override various methods in the each state to respond to events and provide the application functionality necessary for each state to do its job. Obviously in this game, we wouldn't have more than one state active at a time because each state is independent of one another (although X-VP allows you to have multiple states active at the same time and at different hook points).

A typical run of the game might go something like this. Upon application start, after X-VP has been initialized, the Intro Movie State is added as a Pre-Update State to X-VP. In our Intro Movie State, we play a movie and continuously check for user input to skip the movie. At the end of the movie or when user input is detected, the Intro Movie State removes itself from X-VP and adds the Main Menu State as a Pre Update State.

Now, the Main Menu State is activated and begins receiving update notifications in which it handles user interface logic. If it determines that the player wants to start a new game, it removes itself from X-VP and adds the Game Play State as a Pre Update State.

Now, the game is in play and the only state interfacing with X-VP is the Game Play State we last registered. The game play state might be adding and removing nodes from the Scene Graph like models, skin meshes, particle systems, etc. It might be playing audio and checking for user input or any number of other tasks. At some point, the game play will stop (the player dies) and in this particular example we want to go straight back to the main menu. At this point the Game Play State removes itself from X-VP and adds the Main Menu State as a Pre Update State. Now we're back to the main menu.

Obviously state transitions are determined by the application, not X-VP. They can be as simple or as complex as necessary and can add and remove each other according to whatever rules you enforce. Although not shown in this example, it's possible to have more than one state active at a time. For instance, we could have had a seventh debug state that was added to X-VP as a Post Render State. In this Debug State, we could output performance statistics on the screen. Since the Debug State is added as a Post Render State, the debug state will be updated once everything has been updated and rendered.

There are countless ways to use multiple states. You could even have a state that simply takes as input, two other states and performs screen transitions between them.

For more information on the methods available to a state see the X-VP API reference. Also, see the tutorials for example code.

7 Conclusion

The introduction provided in this document has touched on several key areas of X-VP. For further reading, see the X-VP API reference. Also, take a look at the tutorials to see sample applications taking advantage of the concepts introduced in this document.

For help in setting up the SDK with your particular environment, see the Getting Started with XVP Guide.