ZED

Engine Design Document

Open Game Developers

[Version: 0.0.0.2] [Date: 01|10|2012]

ZED

Change Log

Version	Author	Changes
0.0.0.0	Rico	Initial revision
0.0.0.1	Rico	After design whiteboard session [see the image file in Docs/Whiteboard/Design/WhatsInThere.jpg] Added various components after the whiteboard design session Expanded on narratives in Proposed Engine Components section
0.0.0.2	Rico	Expanded upon the design goals of ZED

ZED

Table of Contents

ZED

Preface

This document describes the ZED Engine from a high-level perspective with no API-specific references or pre-conceived notions of what the engine is going to be doing at a low-level or at an external API-level.

General concepts, such as rendering viewports, network connections, or multi-channel speaker configurations will be described from a purely descriptive, non-specific viewpoint. To elaborate, no specific mention of the file descriptors for making network connections, or anything more detailed than the dimensions of a viewport, will be divulged.

Other than for the engine maintainers, this document will not be of much use. A Programmer's Guide document is planned for after the API is designed. For the design of the API, see the API Design Document. There are no programming samples in this document, the Programmer's Guide document, which is forthcoming, would be more appropriate for seeking information on how to use the API.

ZED

Introduction

Executive Summary

ZED shall be designed to allow for developers to create interactive 3D worlds with a high-level and low-level API. Two paradigms are available; object-orientated and procedural, implemented in C++ and C, respectively. The decision to provide two APIs will give the developer a choice between a 'pure' C++ library and to allow for the developer to implement ZED on more processing-power limited hardware. For instance, the SEGA Saturn does not have a lot of room to allow for a class with multiple levels of inheritance and multiple abstract virtual functions; with a 4KiB cache, this is quite a hindrance. Adopting a C and C++ set of libraries enables developers to not have to compromise and attempt to create a similar interface if they want to use a ZED-like façade for their project.

Overview

In making developers lives easier, ZED will be acting like an Operating System's microkernel or a true kernel. There will be core services available and from there modules are appended to allow for a minimal amount of dependencies. This will also allow for developers to extend the engine or use a different set of underlying APIs to implement an alternative library for a specific piece of hardware, for example.

Giving developers an API where they are in control of every path the code can take is an important goal of the engine. If the renderer is too rigid that it can only support Direct3D and OpenGL, then platforms that either implement a different API or cannot support them are instantly locked out of the development effort. This would lead to developers requiring a different engine, building their own, or trying to shoehorn in the rendering library they want to utilise. Using a modular approach to the implementation will enable and encourage a developer community to grow and contribute to the engine's advancement across a wide variety of hardware and platforms.

ZED will provide multiple libraries which will be supported by Open Game Developers. These libraries will encompass debugging, storage, memory management, exploiting processor capabilities, rendering, audio, networking, resource handling, and input handling. In addition to these libraries, there will be artificial intelligence, animation, and physics libraries as well as a stand-alone scripting language.

ZED

Design Goals

ZED should offer developers an easy-to-use, flexible, open, expandable, and modular engine.

It should be possible for developers to be able to get the engine's code and start writing games without having to attempt to accommodate the engine. The engine should accommodate the developer.

Keeping up-to-date documentation with a high degree of integrity is critical to new programmers who are evaluating the engine for integration.

Without too much effort, a new rendering technique or networking model should be easy to iterate on and integrate into the engine for other developers to use if they wish to distribute their code. Ideally, the new engine component should be able to integrate with the main development effort supported by Open Game Developers.

Abstraction is key to moving the API out of the game programmer's way and allowing them to more quickly describe worlds.

Wide variety of platform support:
Personal Computer Operating Systems
Windows [2000, XP, 7]
Linux
Mac OS [10]
SunOS

Home and Handheld Consoles
SEGA Saturn
SEGA Dreamcast
Microsoft Xbox
Nintendo DS
Nintendo GameCube
GP2X
Open Pandora

Tablets
BlackBerry PlayBook

Proposed Engine Components

System

All other libraries use the functionality provided by this component.

Basic hardware is accessed from the System API. This includes the memory, mass storage, and processor. External peripherals and any hardware components that are not critical to the operation of a computer (such as graphics cards and networking interfaces) are excluded from this library. Other hardware is accessed in their respective libraries.

In addition to providing interfaces to hardware; the API will be responsible for managing the hardware's resources, such as how to access memory and files. Making the best use of the processor requires knowing how many processors are available for parallelism and what the capabilities are.

Memory fragmentation issues will also be handled by the System API as well as accessing, reading, writing, and listing files and directories.

Data Types

The Standard Template Library [STL] is concerned with generality over performance. ZED's containers should be much more performance-orientated.

Arrays, lists, queues, sets, and maps will all be replicated to be more performance-orientated than their STL counterparts.

Arithmetic

Mathematical constructs which use the resources of the platform the application is being run on are vital to ensuring that 3D primitives are manipulated as fast as possible and that any intersection tests are performed in a timely fashion. For 3D primitives, there are vectors, matrices, and quaternions which will form the basis of manipulating 3D primitives. Intersections will be performed with rays, lines, planes, frustums, bounding volumes, and polygons.

If there is a SIMD unit available on the processor (information provided by the System library) then It will be exploited to accelerate calculations.

Renderer

Rendering is concerned only with the viewports being rendered into. Any windowing system specific activity is up to the user implementing their project. Fullscreen rendering and the like are not handled directly by ZED, only the rendering context is provided. Of course, some window system specific data types will need to be passed to the renderer so that rendering can occur.

The Renderer will need to scale from hardware on older platforms that doesn't support shaders, up to the latest-and-greatest graphics card that supports native GPGPU. As there are graphics cards with fixed function shaders, there will need to be a fallback for those cards. For programmable function shaders, there will be support for applying and managing them as well as interfaces to set the data they require. In both the programmable and fixed function cases, vertices and textures will need to be managed. The renderer does not have any concept of scene management, animation of any sort, nor models as comprised from meshes. These will be supported from separate components.

Shaders are implemented on a per-graphics capability set, meaning that the user will have to specify the minimum graphics profile available and ensure they offer valid alternative code execution paths for differing profiles. Built-in shaders are provided for rendering lines and basic textured, lit vertices.

Textures will utilise file formats that the graphics device natively supports, such as S3TC, PVRTC, ATITC, ETC, and DXTx/BCx. Online conversion of textures would be ideal for testing textures.

Animation

While animation is related to rendering, it is not included in the Renderer library because it is another layer building upon the core rendering functions. Morphing, skeletal, and per-vertex animation will be supported.

Scene Management

Scene management is intended to be flexible and extensible for different world partitioning systems such as Binary Space Partitioning, Octrees, Quadtrees, and Potential Visibility Sets.

Assets

Vertices can be rendered from a ZED model file. Model files can be static models or animated with skeletal animation, shaders and textures are also referenced in model files.

Fonts are handled either as vector or bitmap fonts. Loading and processing fonts for conversion on-the-fly should be made possible to allow for rapid testing without the need to process the fonts in an offline tool.

Video files will be processed by ZED from Ogg Theora files.

Resource Management

In order to ensure that the mass storage device isn't getting accessed unnecessarily; only one copy of a resource is guaranteed to be loaded into memory at any given time. Resources are unloaded from memory when another resource is requested and there isn't enough space available to handle the resource.

It should be possible to stream resources from a storage device or a network device to allow for assets to be available almost instantaneously. Default assets should be used when an asset is not yet available if possible.

To aid in resource management, an archive format which can be compressed or uncompressed would help in loading resources from storage by just jumping around in one file rather than opening up multiple files which could be a performance issue for hundreds of resources be accessed in a short period of time.

Input

There is a myriad of hardware available for input today and there will be many more input devices in the future. To help abstract the differences in hardware, the input library provides a way to link input actions to a common processable format for games which is defined by the user. As hardware will be vastly different, there should be a common interface for accessing the input device information required by a title.

Devices can provide information relating to digital or analogue buttons, keys, touch screen position, mouse position, absolute position and rotation data, RGB and depth images, and accelerometers.

Audio

Processing compressed and uncompressed audio through multi-channel audio devices. As well as provide playback controls.

Provide volume, reverberation, positioning, panning, and pitch manipulators.

The Open Source file formats Ogg Vorbis and FLAC would be ideal for an audio file format.

Networking

Provides functions to connect remote machines, whether they be on a LAN, MAN, or WAN. Sending and receiving data, and packing and unpacking aforementioned data. Client/Server and Peer-to-Peer connectivity paradigms.

A latency simulator API for LANs is a must if developers are testing in a local environment.

Artificial Intelligence

This library is responsible for imbuing non-player characters with seemingly intelligible actions; steering behaviours, pathfinding, representations of the world for the agent to navigate, decision trees, state machines, and behaviour trees. All of the techniques should be open for users to provide values that change the behaviour of the agent. Blending of different states and behaviours should be possible.

Physics

Rigid-body and soft-body physics as well as collision detection will be handled by this library. Mass-spring and particle physics will be employed.

Scripting

To help get designers creating more interactive worlds without the need to create C or C++ code to fulfil their requirements, a scripting language would benefit both the designers and the programmers. Mitigating compilation time and allowing code to be replaced on-the-fly would be great for AI programmers tweaking parameters or even algorithms prototyped in script then compiled in a native programming language.

Intrinsic Functionality

The functionality that follows here is listed as intrinsic, which means that it will be embedded into all of the components of the engine in one form or another.

Debug Information

Debugging information can come in a variety of forms. Whether it's showing normals of a model or a command-line console, debugging functions are imperative to tracking down errors in the execution of a game. The debug functions will have to draw to the screen as well as communicate over a network for remote debugging communication. Logging to a debug file will also prove helpful for attaching to a bug report, as will a recording of a video of the game in-progress.