sourceforge Welcome, Guest! Log In | Create Account

Visit project openreng

Manual All

Building OpenREng from source

Step-By-Step Instructions

1. Download CMake 2.6 from www.cmake.org

- CMake 2.6 is required.
  - LINUX: Automatic packaging tools may not have this version in their repositories. Download CMake source and build it when required. CMake has no additional lib dependencies. It can build on your platform easily.
  - Windows: You can download installer from CMake.org.
- REng does not distribute project/makefiles, that's what CMake is used for.

2. Check-out REng source from 3dphone svn repository

- You can check out trunk or any branch/tag folder only if you would like to.

3. (Optional) Check-out dependencies folder from 3dphone svn repository

- This folder includes headers and precompiled binaries of the libs.
- The directory structure should be as the following if you want to use given dependency folder structure:
  - {DIR}/dependencies/include
  - {DIR}/dependencies/lib
  - {DIR}/REng/trunk/main (etc)
  - {DIR}/REng/branches/my_branch/main (etc)
- Dependencies are available in repository to make it easy for others to download/build the libs. You can manage the dependencies yourself too, you don't "have to" use those files in the repository.
- The official (maybe modified) versions of the dependencies can be found under packages folder. You can (should) use these source packages to build those libraries on your own platform.

3.1 Make sure that dependencies/lib folder holds the lib binaries

- Extract the pre-compiled lib from the precompiled folder. Lib files should be IN this directory, not subdirectories, etc.
  - UNIX: Copy the .so files into /usr/lib
  - WINDOWS: Copy the dll files into Windows/System32 OR to the application/test bin folders (suggested).
- Remember that you can build the libs yourself and put the binaries here / the system folders as required.

4. Generate your platform/compiler/IDE specific makefiles

- Make sure you configure the settings correctly in this step!

4.1 Read Running CMake

- Want more resources? Try : http://www.cmake.org/cmake/help/cmake2.6docs.html, http://www.cmake.org/Wiki/CMake or [www.google.com google] it.

4.2 Decide whether you want to use OpenGL 3.0> or OpenGL ES 2.0

- You must set USE_OPENGL_ES flag correctly in CMake before you generate the files.
  - FOR OpenGL 3.0>
    - Make sure your drivers support it. Download http://www.realtech-vr.com/glview/ or any other tool (or maybe use your graphics driver) to learn if your computer supports it.
  - FOR OpenGL ES 2.0>
    - Make sure you download an ES Emulator SDK and that their demos WORK. Then install OpenGL ES 2.0 (headers and libs) into your system.

- See [#Installing OpenGL ES 2.0 Emulator on PC's] for detailed instructions for this step.

4.3 Decide where you want to generate the makefiles / intermedita build files

- Where you checked-out the REng is your SOURCE_DIRECTORY. Ex: The directory has the following folders in it: main, apps, tests, lib, etc
- You can set ANY directory for your BUILD_DIRECTORY. (ex: myHome/myBuilds/config1/)
- Prefer not to set your BUILD_DIRECTORY to the SOURCE_DIRECTORY.

4.4 (Optional) If you are compiling on OMAP, set PLATFORM_OMAP to true

- Only available only in UNIX builds.

4.5 Read Running CMake again and generate the makefiles

- Visual Studio
  - The project files hold 4 build options: Release, Debug, etc, etc. These are automatically created common configurations for builds. You can chose any of those you like, but debug and release settings are modified a little bit they work out-of-the-box, if you use other settings, you will need some tweaking...
  - The apps and tests are linked to REng static library build. You may skip building Shared library in default configuration if you would like to.
- UNIX Makefiles
  - If you want to specify debug-release build, you should do so when you run cmake command (see cmake documentation for details).

4.6 Use the makefiles generated by CMake to compile on your platform

Notes:

- The libs/executables are generated in the svn directory, the solution and intermediate files (ex:obj) will be build in your build directory.

Notes

Installing OpenGL ES 2.0 Emulator on PC's

- On OMAP, you do not need to perform this step :)
- There are two HW Emulators for OpenGL ES 2.0:
  - PowerVR (For Linux and Windows)
  - ATI (For Windows Only)
    - Currently, REng is only tested with PowerVR's emulators. ATI's emulators should also work fine, but they are not currently tested.

1. Download an OpenGL ES 2.0 Emulator SDK for your platform (preferably PowerVR's).
2. Extract the emulator to some folder.
3. Install headers
   - Header files are EGL/egl.h , GLES2/gl2.h and other .h files along them. Locate those files in Emulator SDK.
   - WINDOWS VS: Copy them to \Microsoft Visual Studio 8\VC\include folder (one of the default search folders)
   - UNIX: Copy them to /usr/include
4. Install lib binaries
   - WINDOWS VS: Lib binaries files are libEGL.lib and libGLESv2.lib . Locate those files in Emulator SDK.
     - Copy them to \Microsoft Visual Studio 8\VC\lib folder (one of the default search folders)
     - Also, you need to copy corresponding dll's to apps-tests bin folders or system32 folder.
   - UNIX: Copy them to /usr/lib

OpenGL Wrappers

OpenREng includes object-oriented OpenGL wrappers, to provide easy OpenGL server-side object management. This page summarizes available wrappers, the basic theories under the wrapped objects and the specific OpenGL functions which are included/called in the wrapper classes.

General Notes

- You can use the GPU resource abstractions (under REng/GPU folder) separately from OpenREng to aid in you using OpenGL commands.
- GPUDrawer renders "MeshGeoms", thus cannot be separated from the OpenREng system.
- The OpenGL resource ID's are not available to public access, to prevent manipulation OpenGL objects outside the wrappers. If you need the interfaces to be extended, please send requests to developers or send your own patch files.
- You should not call OpenGL functions that have been wrapped outside the REng library.
- The GPU Wrappers also include render properties, which are listed under material system.

GPU Resources

Specific types of resources, that have been wrapped by OpenREng, can be created by the application using OpenGL interface. Each of these resources are assigned a resource ID by OpenGL driver. The wrappers automatically creates these resources on construction and destroy them on destruction, following Resource Acquisition Is Initialization concept. The resource object types are:

- GPUBuffer
- GPUFrameBuffer
- GPUProgram
- GPUShader
- RenderTargets
  - GPUTexture
  - GPURenderBuffer

GPUBuffer

The buffers provide a generic memory abstraction. Two generic classes of buffers exists within OpenREng: SWBuffer and GPUBuffer.

SWBuffers are used to create/reference/modify continuous memory that resides in main system memory. It merely wraps over malloc/free/memcopy calls.

SWBuffers are used to create/reference/modify continuous memory that resides in GPU memory. GPUBuffer wrapper mainly supports all GPU-side buffer related functionality in OpenGL ES 2.0, yet does not currently support all the features of the buffer extensions found in desktop counterparts.

Specialized vertex and index buffers exists, which provides interfaces assuming all their data is homogeneous given the data properties. These buffers can be stored in GPU or main system memory, depending on their parent in inheritance hierarchy (SWBuffer or GPUBuffer).

Wrapped GL functions:

- *glGenBuffers, glDeleteBuffers, glBindBuffer, glBufferData, glBufferSubData*

OpenGL 3.0 and above only:

- *glMapBuffer, glMapBufferRange, glUnmapBuffer*

GPUFrameBuffer

OpenGL renders into (and reads values from) a framebuffer. GL defines two classes of frame-buffers: window system-provided and application-created. This class provides an abstraction over "frame-buffer object" operations. The framebuffer-attachable objects are RenderTargets (GPUTexture or GPURenderBuffer).

By allowing a rendertarget to be attached to a framebuffer, the OpenGL provides a mechanism to support off-screen rendering. Further, by allowing the images of a texture to be attached to a framebuffer, the OpenGL provides a mechanism to support render to texture.

Notes:

- Frame-buffer bind target is implemented as a global state, to make interfaces simpler. This state can be implemented as per object as well (? TODO)
- REng framebuffer stores the depth, stencil and color attachment information for future use.
- Attaching to framebuffer is currently done using render target objects and using the global "active framebuffer" object and thus is state base. The active framebuffer is notified of attachment on success. (TODO: change and favour a frame-buffer oriented approach?)

Wrapped GL functions:

- *glGenFramebuffers, glDeleteFramebuffers, glBindFramebuffer, glCheckFramebufferStatus*

GPUTexture

Textures are 1D, 2D, 3D or cube-map images that can be loaded to HW and be used as texturing resources. (1D and 3D is not supported in core OpenGL ES 2.0 specifications.)

This class is a wrapper over most OpenGL texture related operations. The aim is supporting for all internal formats available (and managing platform-dependent types automatically)

- The wrap-mode, minification and magnification texture parameters are specified using Texture Render Properties.
- You can request **automated mip-map generation** on texture upload time.
- Cube-maps can store depth component textures. (ex:rendering shadow maps for a point light)
- To be able to successfully load and use a cubemap texture: The level zero arrays of each of the six texture images making up the cube map must have identical, positive, and square dimensions.

Attaching to frame buffer:

- GPUTexture class allows a texture to be attached to active frame buffer. A GPUTexture object automatically tries to attach to correct depth / stencil / color0 buffer depending on the image format. To attach to a specific color component, you have to provide the color target in the related methods.

Wrapped GL functions:

- *glGenTextures, glDeleteTextures, glBindTexture, glGenerateMipmap*
- *glTexImage2D / glTexImage1D / glTexImage3D*
- *glFramebufferTexture1D / glFramebufferTexture2D / glFramebufferTexture3D*

GPURenderBuffer

A render buffer is data storage object containing a single image of a renderable internal format. A render buffer is a render target (it can be attached to frame buffer and store render data).

Note: A texture cannot support multi-sampling, but a render buffer can. Yet, a render buffer cannot be used directly for sampling a texture in shaders.

Attaching to frame buffer:

- GPURenderBuffer class allows a texture to be attached to active frame buffer. A GPURenderBuffer object automatically tries to attach to correct depth / stencil / color0 buffer depending on the image format. To attach to a specific color component, you have to provide the color target in the related methods.

Wrapped GL functions:

- *glGenRenderbuffers, glDeleteRenderbuffers, glBindRenderbuffer*
- *glRenderbufferStorage, glFramebufferRenderbuffer*
- *glGetRenderbufferParameteriv (with param GL_RENDERBUFFER_SAMPLES, GL_RENDERBUFFER_RED_SIZE, etc)*

GPUShader

A convenience class for GLSL shaders.

Wrapped GL functions:

- *glCreateShader, glDeleteShader, glShaderSource, glCompileShader*
- *glGetShaderiv (with GL_INFO_LOG_LENGTH, GL_COMPILE_STATUS parameters), glGetShaderInfoLog*

GPUProgram

This class encapsulates the features of OpenGL Shading Language programs.

Notes:

- Multiple shaders of the same type can be attached to a program if not using OpenGL ES 2.0 configuration.
- OpenGL info logs are written into material system log files and are not stored internally persistantly.

Wrapped GL functions:

- *glCreateProgram, glDeleteProgram, glUseProgram*
- *glAttachShader, glDetachShader*
- *glLinkProgram, glValidateProgram, glGetProgramiv with GL_ACTIVE_ATTRIBUTES, GL_INFO_LOG_LENGTH, GL_LINK_STATUS parameters*
- *glGetActiveUniform, glGetActiveAttribute, glBindAttibLocation*

OpenGL 3.0 and above only:

- *glGetFragDataLocation, glBindFragDataLocation*

GPUUniform

It is a convenience class for OpenGL GLSL uniform variables. It stores uniform name (string), uniform resource location (int) and the HW program the uniform belongs to.

Wrapped GL functions:

- glGetUniformLocation, glUniform*

GPUDrawer

GPUDrawer is aimed to draw REng-specific mesh geometries (which hold vertex and index data) to active frame buffers using the active program. It is aimed to allow batching of mesh geometries and disabling binding of vertex/index buffers when necessary. The API is not yet complete and is subject to change to allow faster draw calls.

Wrapped GL functions:

- *glDrawElements, glDrawArrays*
- *glEnableVertexAttribArray, glDisableVertexAttribArray*
- *glVertexAttribPointer*
- *glMultiDrawElements, glMultiDrawArrays (TODO)*

TODO

- Merge GPUUniform and RenderProperty_Uniform, to provide a simpler interface ?
- Extend the support for recent OpenGL Desktop extensions related to GPUBuffers.

Material System

A material in real world is used to describe how an object looks, and behaves. A velvet fabric has specific reflection and refraction (BRDF) properties, has a color/texture and feels different from a satin fabric when touched. The material system in OpenREng is used to describe how a virtual 3D mesh model is to be rendered using local illumination. More technically, a material provides an abstraction over the rendering state when a model is rendered to screen.

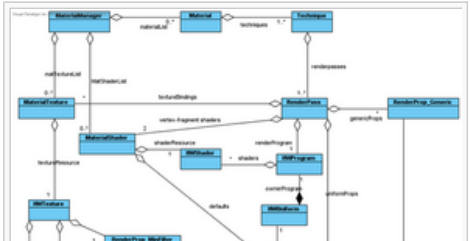
Summary

OpenREng material system supports

- Multiple-pass rendering
- Multiple-techniques per material, indexed through distance and view ID's at runtime
- Easy management of OpenGL rendering states
- Optimizations for redundant state changes
- Material scripts:
  - Advanced material scripting using text files, exposing most underlying functionality
  - Resource management and delayed loading of resources
  - Automatic platform related capability management (between desktop and mobile) when required.

Overview

The following class diagram shows the overview of the material system implementation:



Brief description of the components involved:

- **MaterialManager:** Objects in the material system can only be created by and requested from the MaterialManager, it maintains unique identifiers.
- **MaterialShader:** Adds naming, source file definition and uniform defaults definition features over basic GPUShader. It can hold uniform defaults to be activated when the shader is linked to programs.
- **MaterialTexture:** Adds naming, source file definition and importing image files features over basic GPUTexture.

- **MaterialProgram:** Adds naming feature over basic GPUProgram, material shaders can be attached by material names and resolved in link time. It can hold uniform defaults to be activated when the program is linked.
- **Material:** Stores the techniques the material has and allows accessing the most suitable technique for the given lod / view configuration
- **Technique:** Stores a list of render passes (for multi-pass and multi-view rendering)
- **RenderPass:** A collection of GL program(vertex shader-fragment shaders) + uniform + generic + texture binding states. The basic logic to render a mesh is : prepareState(); drawmeshgeom(); clearState();
- **Render Property:** See below.


Material class relations (Old)

Because of design requirements, the material abstraction described above is based on a "render-pass" definition. Complete shading of an object may require rendering of the object multiple times (as in non-photorealistic techniques or when the hardware resources or API specifications (limitations) cannot support a rendering technique in one pass). Also, optimizations based on specifying different material properties for different distance and multi-view configurations is also possible and has a high positive impact on the overall efficiency and design. This allows specifying different rendering techniques for different view and LoD configurations. It is highly recommended to specify simpler rendering techniques for objects that occupy a small space in screen (ex: when an object is farther from the camera in a perspective projection) and separate views, using the perception-based cues when possible.

The material properties are based on OpenGL specifications. All the basic parts of the rendering pipeline that affect how an object is rendered is represented in OpenREng and exposed through material scripting.

The materials, textures and shaders are indexed through unique string names. Thus, the content pipeline is based on using names as indexes. The 3D meshes store a material reference and mesh files describe the material of a mesh by specifying the material name. A material definition in a material script is required to reference shaders and textures, and this is also based on indexing through unique string names of shaders, textures and programs.

Render Property

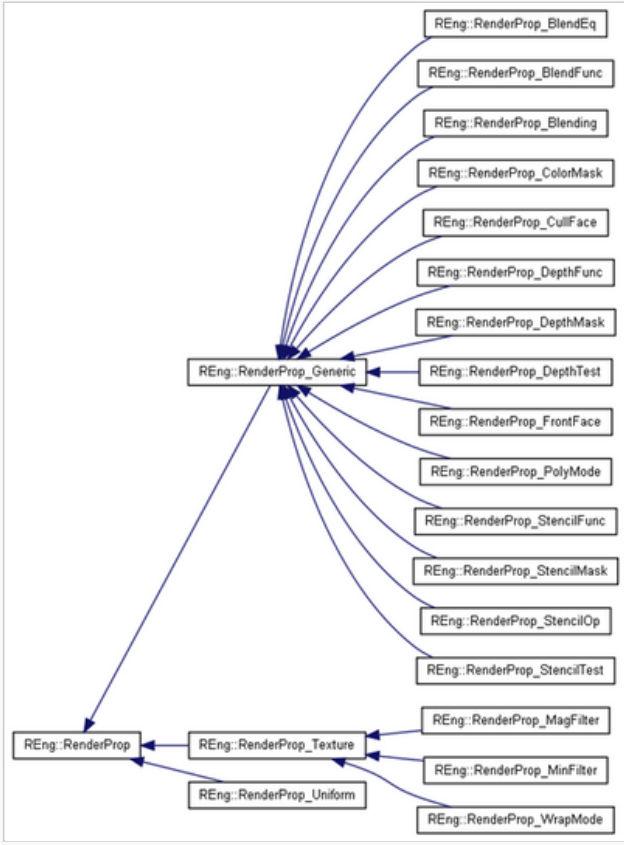
A render property aims to provide an abstraction over basic OpenGL shading states, allowing easy material-script based definitions, state tracking and optimizations.

As can be seen in class diagram, there are three types of render properties:

- **Generic property:** Wraps over an OpenGL state description that is related to fixed-function parts of the pipeline.
- **Uniform property:** Wraps over customized shading parameter (uniform) description (Commonly stored inside material shader & render passes)
- **Texture property:** Wraps over a texture parameter (Commonly stored inside material textures)

To define a render property, you can either use material scripts to associate a render property with a render pass / texture / shader or you can manually create and insert render properties to related data structures through C++ API. The OpenREng renderer automatically updates the related properties, when a render pass is activated or a texture is loaded. For customized use of a render property, the render property API provides *activate* and *deactivate* methods. Deactivate methods are only functional for render properties that have a "default" state, as described in related internal project documentations.

Applications should use available render property wrappers in OpenREng for updating basic states (blending, stencil, depth and face modes). This follows the basic approach which prohibits the use of OpenGL commands outside the OpenREng library and allows updating OpenGL states through OpenREng commands.



Class diagram of available render properties

State Tracking And Optimizations

The render property abstraction also allows controlling and reducing the number of state changes automatically inside OpenREng. The advantages can be seen clearly in uniform property usage. The uniform property stores the value of the uniform the programmer wants it to be active. When a material uniform data is updated, it is set dirty. When the shading program that the uniform is bound to is activated, the dirty uniforms are uploaded to graphics hardware and thus synchronization is provided only when required.

The texture properties are similar to uniform properties; they are attached to textures instead of shading programs and are synchronized on texture upload time and when the texture is bound to a render program.

The generic properties reflect a global OpenGL server state. Tracking currently results in following behavior: If a generic state has a default value and render property is set to the default value, activate/deactivate does not change the render state. Currently, the render states that support tracking are:

- Depth function, Depth mask, Color mask, Front face, Cull face, Face mode, Polygon mode, Depth test, Stencil test, Blending (on/off)

Since checking for current state before updating a state has an additional computational load, and each state change may stall the rendering pipeline by a different time, enabling and disabling state change optimizations can be enabled/disabled on compile time and also on run-time. Compile-time disabling completely removes any state tracking computation and thus overhead in render properties.

Platform Capability Management

Since mobile/embedded OpenGL API is slightly more restricted than the desktop OpenGL API, a material property may not be available in a mobile environment. An example is the polygon mode, which sets how polygons are rendered, and the possible options are filled, line and point rendering. In OpenGL ES 2.0, this fixed-function state is not configurable and is fixed to FILL. Thus, **polygon mode property has no effect on**

OpenGL ES 2.0 configurations. This cases is automatically detected, skipped if possible and warning information is generated to log files and returned through API, for informing the application programmer about the incompatibility of a material definition.

OpenREng is based on OpenGL specification, it expects the hardware to conform them. OpenREng does not make any use OpenGL extensions, thus it is currently aimed to work on core OpenGL specifications and does not detect additional extension-specific behavior.

Material Scripting

The material system in OpenREng is also exposed through material script files, following a data-driven approach. The script file format is a customized and simple format. OpenREng includes a material parser that generates run-time material data from a given material file. When a syntax error is found in a material file, the remaining file is not processed.

A material file can declare the following data:

- The vertex / fragment shaders (with usability extensions over regular OpenGL)
- Textures (loading from files, setting texture properties)
- Programs
- Materials

Check the sample.material file within the source distribution (in folder apps/bin/materials) for detailed description of complete material file syntax.

An important property of the material system is that you do not have to define an indexed texture or shader before using it. The links are resolved when links are referenced. The most important application of this approach is when loading textures. A texture may be declared in a material file and an internal texture may be created, but the texture data (the pixel values) are not loaded from the file and uploaded to the graphics hardware until they are referenced.

Notes:

- The parser generates a separate log file in parsing steps: logs/matParser.log (you can configure log.properties file to change that)
- Check the log file when you update any material file that is loaded. The system is highly likely to be unstable / can crash if the material file cannot be parsed completely or a material could not be loaded because of shader / image loading errors. (Check for any ERROR in log files especially.)

TODO

Update Material Scripting

- Support pre-defining fragment output data bindings.
- Use a scripting language and interpreter (Lua) for material parsing / definitions.
- Extend functionality (frame buffer control, etc) / scene description (Links to scenegraph concept!).

Mesh

The aim of OpenREng mesh subsystem is to describe flexible and efficient model geometry representation which can be easily extended to support advanced features. OpenREng mesh representation is composed of LOD'able mesh geometry definitions and a rendering method specification (through materials). Since OpenREng is built on top of OpenGL and since the basic approach is to not put additional restrictions on top of OpenGL, a scalable mesh structure is defined and extended with functions that allows easy 3D object definitions.

Summary

- A mesh in OpenREng is designed as a collection of renderable per-vertex data that can be rendered in a single draw-call in OpenGL.
- OpenREng mesh supports highly customizable per-vertex LoD definitions.
- A mesh uses a single material, which is activated when the mesh is to be rendered.
- A custom attribute definitions are supported.
- Pre-defined semantics are introduced so that multiple shaders and 3d objects can work seamlessly without too much work done by the application programmer and model designer.

Specifying Vertex Data in OpenGL

The advanced features and limitations of OpenGL that directly relates to specifying vertex data is described in this section.

As noted in OpenGL specifications, a mesh data can either be stored in client's address space or the server's address space, which can be used internally in the GPU. The server address space is managed by using OpenGL buffer functionality. Basically, the buffers are external memory blocks that can be requested and used by the application and there are OpenGL binding targets to buffers that are used for certain operations, such as drawing. In drawing geometry, if data is stored on client side, this data is send to server (GPU) in every draw call.

[[File::class_hwbuf.png]]

Note that OpenGL 3.1 extends OpenGL buffer object bind targets to support other advanced functions, such as copy read buffers, copy write buffers, pixel pack/unpack buffers, texture buffers, transform feedback buffers and uniform buffers. Those buffer types are not supported in OpenGL ES 2.0 and currently are not specified and used by OpenREng, although abstractions for those buffer types are available. These buffer types may be defined as the features are enriched for OpenGL 3.0 and above as required.

Vertex Attribute Semantics Concept OpenGL ES 2.0 and OpenGL 3.1 removes attribute semantics, which was available in previous desktop OpenGL API's. The attribute semantics are used to map individual pre-defined vertex components to shader variables, as there are types of semantics that many shaders/models share. Some important example semantics are vertex position, vertex normal, vertex texture coordinate (or coordinates). New OpenGL pipeline uses ASCII strings to name a vertex attribute and integer-based shader locations of the named attributes are supplied to the graphics application client (which is OpenREng in this case) which can then be used to manage vertex attributes inside OpenREng.

Vertex Declaration Concept OpenGL does not have vertex declaration concept and this makes binding vertex attributes a responsibility of a rendering engine developed on top of OpenGL. It is necessary to bind vertex attributes stored in one or more vertex buffers to a shader object. A vertex declaration defines where the vertex data is and how attributes should be retrieved from buffers. It is important to note that vertex attributes and their position in buffers can change per model. Models can be defined with a variety of vertex types in any order. So a hard-coded solution, where vertex types are statically linked to buffer positions, is not suitable for a generic rendering engine.

Architecture : Mesh Data

A 3D mesh is composed of the following geometry/render data:

- Geometry Data : Geometry data is stored as a LoDed data. Different LoD levels can share vertex data, share vertex and index data (ex: with some other mesh), or can define their own vertex/index data.
  - Vertex data
    - Buffer element range : This range specifies the -shared- range in all vertex buffers which the vertex elements are stored.
    - Vertex-buffer list: This list stores shared pointers to vertex buffer objects. OpenGL or client-side buffers can be shared between different vertex data objects.
    - Vertex attribute list : The vertex attribute data is described below.
  - Index data
    - Buffer element range : This range specifies the range in index buffer which the index elements are stored.
    - Index-buffer : This is a shared pointers to an index buffer object. OpenGL or client-side buffers can be shared between different index data objects.
- Material: This holds the rendering method of the mesh. A shared pointer is used so that multiple meshes can use the same material easily.

The following information lists some of the mesh architecture properties:

- The mesh description is tightly integrated to Material System as seen above.
- A vertex or index buffer is either a client-side buffer or an server-side (OpenGL) buffer. Client-side buffers can be used when frequent updates to buffer data is necessary.
- A mesh geometry (LoDed) may use multiple vertex buffers but only a single index buffer. The index buffer can be omitted to be able to render the vertex data found in vertex buffers sequentially without altering indexes on render-time. This approach follows OpenGL specification.
- Multiple meshes can share the same index or vertex buffer, using different ranges and the same index/vertex data type.
- The meshes do not directly draw themselves. There exists a single module in the system that can draw a given render data and is called the GPUDrawer. This allows optimizing draw calls and obtaining render statistics.

Vertex Attribute Definition

A vertex attribute is composed of the following information (following OpenGL specification):

- *Source buffer index:* Allows separating buffers of attribute semantics of a single 3D mesh.
- *Start index (stride):* The offset in the buffer that this element starts at
- *Data type:* The type of data (floating point, integer, short integer, etc) of the vertex attribute
- *Data count:* The component count of per-vertex data. Example: A vertex position generally requires 3 components.
- *Semantic:* The meaning of the attribute and it is used for handling some types of vertex attributes automatically in shaders.
- *Name:* It is the string name of the attribute as used in a GLSL shader. The pre-defined semantics fill in this name automatically. Otherwise, you have to provide a name yourself.

Vertex Attribute Binding to Shaders

Pre-defined semantically mapped attribute names are defined in OpenREng (for easy use in shaders). The pre-defined names for use in shaders start with "re_" prefix. Examples of semantically mapped attributes are currently "re_Position", "re_Normal" and "re_BiNormal". This follows the OpenGL approach which is available in desktop OpenGL versions before 3.1.

The shader attribute data of GLSL programs is received in OpenREng runtime using the OpenGL driver, after successful linkage of attached shaders into the GLSL Program. The attribute data is stored inside GPUProgram object and later used to be able retrieve attribute location given an attribute name. Note that even if the attribute has the same name in a GLSL program, it may be assigned a different location for use.

Loading Meshes From File

A data-driven 3d mesh approach is highly required to be able to create models in external tools and to use models with the engine to be able to render to screen. The mesh files need to describe all mesh-related information, including vertex data, index data and material. The material script files, which are text-based, are separated from mesh files, to be able to share materials between different objects and also to make it easy to edit material script files.

The MeshManager is used as a mesh loader component. The mesh manager supports **loading meshes from files** and **retrieving loaded meshes**.

The meshes that are loaded are indexed through unique string names inside the mesh manager. Thus, multiple meshes cannot share the same name and loading (parsing) a mesh multiple times is also prevented.

Currently, only **3ds** file format is supported by lib3ds parser, but multiple mesh types can be supported in the future by extending the loading / mesh parser facilities. The file extensions are used to detect the mesh file format. The future plan is to write a mesh format optimized for OpenREng mesh description and implement an exporter for Blender modeling tool.

About Loading 3DS Meshes

Currently used 3ds file format is one of the formats specified by 3D Studio Max and is a commonly used mesh format for exchanging and loading 3D meshes. Using specialized chunks with headers, it can describe a complete scene, with lighting and camera information and store multiple meshes. To be able to fit into OpenREng mesh architecture, some limitations and minor adjustments are applied, as stated in the following list:

- Only mesh data in a 3ds file is processed.
- Vertex position is always loaded and is always composed of 3-component floating points.
- Texture coordinates are only loaded if the mesh stores texture coordinates. Texture coordinates are always 2-component floating points.
- The normal for each vertex is generated using the vertex face data only if the mesh loader setting "mGenerateNormals" is true.
- All the vertex attributes available are loaded into separate vertex buffers.
- The vertices are specified using triangle lists, as stored in 3ds files. No automatic convertion to other triangle format is supported.
- Mesh string names, as stored in 3ds files, are used as names of loaded meshes. If a mesh with no specifed name is found in a 3ds file, the name of the 3ds file is used as the mesh name. Note that mesh names are unique and you cannot re-load a mesh with a loaded mesh name.
- Only the "material name" field of the mesh material chunks in 3ds is used, since OpenREng uses its own OpenGL-centric material property definitions, which can be indexed by using unique material names.

Discussions

Vertex attribute binding - design option 2:

- Define vertex attributes (names and types) in material files.
  - Con: You need to declare the same data in more than one place.
  - Pro: It can also allow linking any attribute to a pre-defined attribute semantic.
    - The shader may use myVertex as vertex position attribute, than define "re_Vertex = myVertex" type of stuff in material files. (Introduces slight data management overhead)
    - Loading attribute definitions to vertex shader objects would be required.

Vertex attribute binding - design option 3:

- Registering attributes before link time, or retrieving attribute locations after link-time.
  - Pre-defined locations for pre-defined semantics.
    - Con : Does not fit well into restricted generic index counts, or customizability?

TODO

- 3dPhone mesh file format.
  - Define file format
  - Implement exporter for Blender.
- Support key-frame animations for a mesh.
- Support skeleton - mesh skinning.
- Support one of the following file formats : md2 / md3 / md5
- Use LoDed mesh geometries in some applications and observe further requirements.
- Study and extend (Sub-class efficiently) usable buffer types for advanced desktop graphics support
  - !BufferBindTarget_Copy_Read
  - !BufferBindTarget_Copy_Write
  - !BufferBindTarget_PixelPack
  - !BufferBindTarget_PixelUnpack
  - !BufferBindTarget_Texture
  - !BufferBindTarget_TransformFeedback
  - !BufferBindTarget_Uniform
  - !BufferBindTarget_None

Scene Graph

Internal management of scene data is a requirement, because brute-force rendering of 3d objects produces low frame rates, where the most of the time is spent on objects that are not visible in the scene. Hidden surface removal (HSR), occlusion culling and view frustum culling are therefore important problems in computer graphics. An efficient method that eliminates 3D objects (meshes) or even triangles from rendering (at some stage) increase the overall efficiency of rendering, thus the rendering engine itself. Another important advantage of using a scene management technique is that it can also be used for faster -approximate- collision detection in a 3D scene.

There is more than one option to manage the scene data. The type of the 3D scene plays an important factor on choosing the management technique to use. For static and indoor-closed environments, BSP's are favoured generally. Quadtrees and octrees are also important data structures that can store scene data. Potentially Visible Set (VPS) structure is another basic scene management technique which identifies visible set of objects from possible camera position regions as a preprocessing stage and use this potentially visible object lists, decreases the number of objects that will be rendered. Portal-based data hierarchies are also very common. Portals are not tree-based, but they basically split the scene into convex (or in some variants even concave) cells, store adjacency information between cells (interestingly, the adjacencies do not have to be physically based) and use the portals and connectivity information to determine which cells are to be rendered. The portal approach may be used to achieve zero overdraw when implemented in a software renderer completely.

Focusing on Scene Graph Structure

The problems that a scene graph structure tries to solve are:

- Provide a database of scene objects, which will be used for both querying and updating.
- Specify the relations between objects.
- Manage scene-data hierarchically.

The scene-graph is mostly used to describe spatial relations between objects. Hierarchical materials can be described as well, by assigning some material property to a node that will affect all the children of that node. This technique has been used to decrease the number state changes, but with recent hardware, state-block updates are favoured against incremental updates.

The disadvantages-limitations with a scene-graph are the following:

- The real-world is not structured as a scene-graph, the entities themselves are part of the environment and interact with each other. If the objects in the scene are physically simulated, there can be no static or pre-defined spatial relation between objects, since the objects are controlled by a physical simulation, not a scene-graph.
- A scene-graph cannot be used directly to support many general graphics algorithms. Many algorithms, such as generating shadows, lighting, proximity queries and applying skeletal animations require their own data structures which may not fit into a scene graph implementation and may require workarounds to integrate into the basic database of objects, the scene graph.

The advantages of a scene-graph approach are:

- It allows simple placement of objects into the scene and also allows sharing of resources easily.
- The objects in a scene-graph can be dynamic and no pre-computed data of a scene is required.

Some more detailed discussions about scene graph structures can be found in:

- http://home.comcast.net/~tom_forsyth/blog.wiki.html#Scene%20Graphs%20-%20just%20say%20no
- http://www.realityprime.com/articles/scenegraphs-past-present-and-future

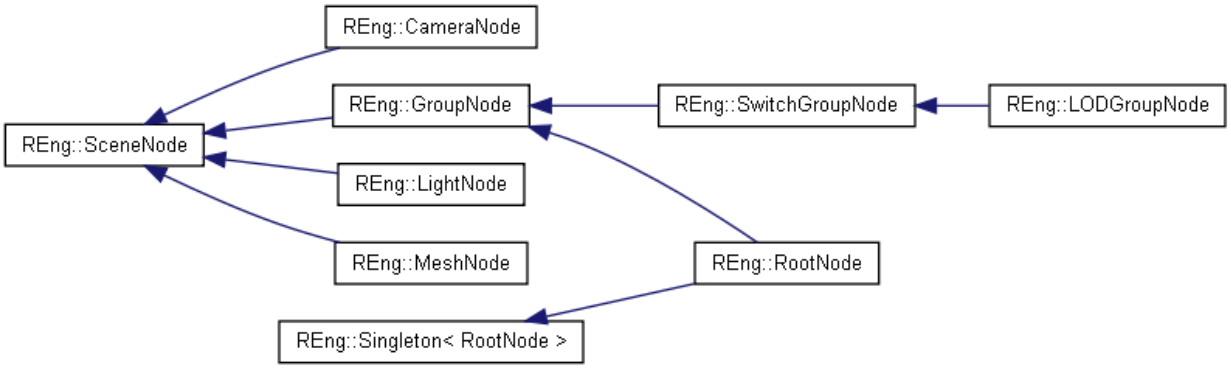
Since the aim of 3DPHONE project is to provide a 3D UI widget based highly dynamic applications (with mostly simple 3D objects) and not on virtual environments with specific properties, the above limitations are not a problem in our application domain.

Design Guidelines

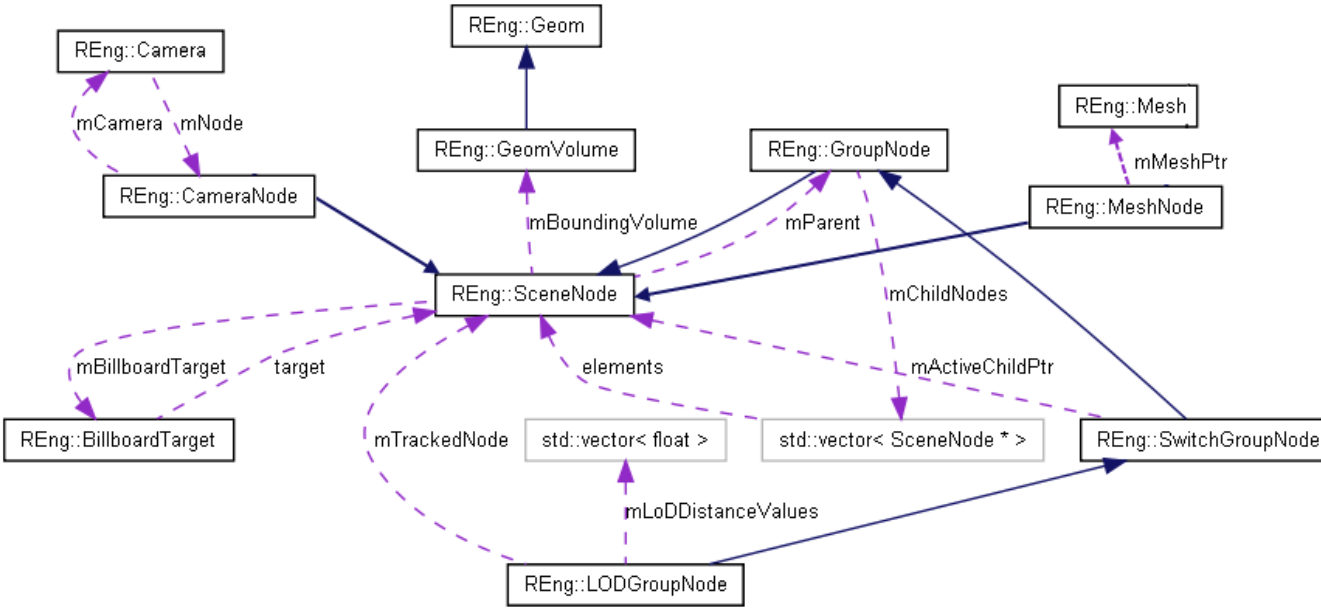
- The scene graph management in REng is basically approached as a database of objects, representing objects locations and orientations and also is used to describe some hierarchy of scene data when found to be useful.
- The previous scene graph implementation is revised to store the separated world-space translation, rotation and scale components as well as the local versions of them. This allows retrieving each component separately and makes the management of billboarding easier. The methods that updated spatial relationships are also reduced to a single method per component with increased parameters, thus simplifying the interface and helping the API programmer understand possible update types.
- The scene-graph API does not allow separating a scene-graph tree into disconnected trees or creating cyclic dependencies between GroupNodes. This is aimed to provide a more reliable scenegraph implementation that may prevent the application programmer to create problematic hierarchies.
  - Currently, you cannot detach a mesh-node from the scene-graph tree (in purpose). It always must have a parent.
- **Construction** : To create a node, you must specify the parent object.
  - You cannot create a scene-graph object on the stack, you must create it on the heap (use create() methods) (Currently, in purpose)
- **Destruction:** : If you destroy a group node explicity, all the children of it is automatically destroyed.
  - Support extended relationships between parent-child node pairs.
  - **Bounding Volumes** : The bounding volumes can support hierarchical auto-managed bounding volumes and multiple bounding volume geometries.

Scene Graph Node Types

The following diagram shows the basic node class hierarchy of the REng scene-graph implementation.



The following diagram shows the overview of the REng scene-graph implementation.



The nodes are uniquely indexed though integer ID's, which can be used in serialization to support cyclic dependencies between nodes. A sample cyclic dependency can be seen in billboard nodes (TODO detailed scenegraph diagram). Also, each node has one parent, except the root node as described below.

SceneNode

It provides an abstract class for nodes. The sub-classes define their own data and operations on this data associated with the node. Most of the functionality is provided by the SceneNode structure itself, such as inheritance properties, culling mode, bounding volumes and spatial updates.

GroupNode

It is a node that can hold a list of children nodes of any type.

RootNode

This is a group node which has no parent and is used as the -single- root of the scene-graph. There can only be one instance of this class in the system, thus only a single scene-graph.

SwitchGroupNode

It is a specialized group node which allows a child to be selected/activated among the list of all children.

LoDGroupNode

It is a specialized group node where all the children nodes must be a MeshNode and the active node is selected based on distance of this node to another tracked node, which defines a position in 3D space.

CameraNode

It is a node that holds a camera object. It allows representing the position and direction of a camera (extrinsic parameters). See Section TODO for camera deign in REng.

LightNode

It is a node that holds a light object. It allows representing the position and direction of a light and also will help to define a hierarchical lighting of the scene.

MeshNode

It is a node that holds a -shared- mesh object. The mesh nodes are currently the only nodes that can store object geometry of the scene.

The types of nodes will be extended in the future as required. Possible extensions are TextNode, which will hold a 3D text, and ParticleNode, which will hold a particle mesh. Both proposed nodes holds scene-geometry as a part of their definition.

Bounding Volumes

The bounding boxes of the objects in the scene can be used for approximate object intersection tests. A complete tri-mesh intersection test in CPU is not a suitable approach, since it involves many operations that cannot be performed in real-time for even medium-scaled triangular meshes.

The bounding volumes of objects can be grouped hierarchically and this structure can be used increase performance, so that using a group bounding box, it is possible to completely pass or completely fail an intersection / inclusion test, which can be used to cull large number of invisible objects in a scene. Note that the bounding volume hiearchy needs to be updated when an leaf bounding box is updated. REng is structured so that the application developer can set or unset generation of a bounding box hiearchy per node. Just as a minor note, we should mention that by using scene-graph approach with bounding volume hierarchies, we can achieve non-regular octree-based or quadtree-based data structures.

Picking

An important use of bounding boxes are for picking. Using meshes for intersection(similarly picking) is time consuming thus an intersection test is performed using the bounding box of a mesh. Each time a click occurs a GeomRay is generated for the clicked coordinate w.r.t. the active camera. The ray then is goes through a number of intersection tests on the Mesh hierarchy starting from the root node. The nearest intersecting MeshNode is found on the mesh hierarchy.

Billboards

- TODO : Specify billboard requirements!
- A billboard node is a node (of any type) that can target and look at a specific node.
- The orientation of the node is updated using the current position of the node when the node transformation is updated.
- Does a billboard orientation update of a node affect the children's orientation's => Should be no.
- Check this page: http://www.lighthouse3d.com/opengl/billboarding/

Extending Node Relations

The relations are specified as inheritance properties, since the relations are based on whether a child inherits a parent node's specific property. These properties are now defined as:

- Spatial translation,
- Spatial rotation,
- Spatial scale,
- Billboard target

These types can be extended as seen required. Note that traversal and update of the scene-nodes using inherited properties must be as optimal as possible, with few numbers of operations on critical execution paths (such as traversing nodes for rendering).

There can also exist relations between objects in scene which are not directly connected in the scene graph. These types of relations cannot be hierarchically defined using the single scene graph instance as found REng. The extension to node relationship as described above allows simple holistic management of 3D scene based only on scene graph when possible.

Loading and Saving

Note: This feature is not implemented yet.

The scene graph hierarchy allows easy streaming of current graph state into an external source (ex: file) To save the basic scene graph, which is a tree, signalling the root node of the scene graph to save a dump out of its content is sufficient. The nodes can recursively generate state information and append this to the output as required. Other objects, such as cameras and lights are serialized and linked separately. The unique node ID's are used to solve relationships between nodes after the node data is loaded and special relationships are to be applied.

TODO

- **Load/Save:** Implement scene data serialization
- **Billboarding:** Improve billboarding requirements and complete implementation of billboard support
- **Inheritance behaviors:** Define different inheritance behaviors (Ex: inherit translate without applying parent cale/rotation)
- Check out X3DToolkit's design and extend it as required.

Camera

A camera is an object which can take a shot of a 3D scene. OpenREng RenderSystem's renderScene method requires a camera object to be

provided, as expected.

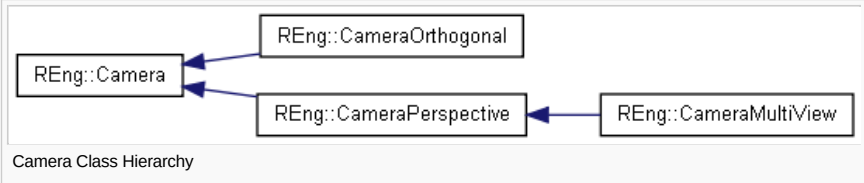
Design

A camera object in REng stores intrinsic camera parameters (such as angle of view, aspect ratio and far distance) and provides an abstraction over real-world cameras. The extrinsic parameters (rotation and translation) are set by the camera scene node that the camera object is bound to. A camera object *always* links to a single camera scene node, so that the extrinsic parameters of a camera are always available on request.

It should be noted that:

- Animating camera position/orientation is based on regular scene graph node animation.
- The "projection" matrix is specified/calculated by a camera object. (see [wiki:REng/Misc#RenderMatrixManager RenderMatrixManager for additional info])

Camera types are sub-classed based on projection properties. A perspective camera is set to provide a perspective projection and an orthographic camera provides an orthographic projection.



Camera Class Hierarchy

In 3D scene rendering, the camera is assumed to be a perfect pinhole camera, with no lens and zero aperture. The current camera architecture does not extend these definitions, but this definition may be extended when a special effect that needs updating camera parameters is required, such as a depth of field effect.

Another highly useful feature that should be provided with a camera implementation is to generate a ray (usually called a pick ray), starting from the viewing position and passing through a given coordinate on 2D projection. Since it depends on the camera position and projection parameters, each sub-class is required to implement its own pick-ray generator.

Multi View Camera

TODO: Link to multi-view section.

Lighting

Using modern OpenGL specifications, you can specify any type of lighting parameters and how a light type affects a rendered object yourself. But OpenGL offers no standard light types and a shader system that can "recognize" current lighting information from the application side. OpenREng provides such a lighting functioanlity, while offering default light types (and shading models within the lights demo).

Basic types of lighting objects are defined and managed by the OpenREng based on common real-time standard lighting methods, and OpenREng allows shader programmers to access lighting information of a mesh easily.

Basic observations/ideas for the lighting system

- A mesh can be illuminated by multiple lights simultaneously.
- The assigned lights can be accessed by low-level shading programs.
- All the lights assigned to a mesh may not be used by the shading program, or a light info required by a shading program may not be assigned by the engine, because such a light does not illuminate an object.
- A light can define a bounding illumination volume and a light culling can be applied.
- The order of lights assigned to an object may or may not be pre-defined.
  - Not pre-defined : A shader cannot "assume" that light1 is point light and light2 is directional light.
  - Partially defined : If light types are separated, shader can apply the same lighting model calculations to each of these lights of the same type.

Light Types in OpenREng

In OpengREng, a light object represents a physical lamb object. "Light_Base" is the basic class type used for a light object. A light is always attached to a scene node and can be lit (enabled) or not. A light object cannot be instantiated, but classes inherited can be. This basic design follows the design for camera structure.

Although you can extend the lighting classes as you would like (see below), REng provides three "standard" light types:

Sun Light:

- A sun light is directional, is not positional and cannot have any attenuation.
- The direction specifies the direction of light rays emitted.
- The sun light has a single color.
- It does not store attenuation information.

Point Light:

- A point light is positional, is not direction and can have positional attenuation.
- The position specifies the source of light rays emitted.
- The point light has a single color.
- It stores light attenuation information. The quadric attenuation factor for a light beam that reaches a point "D" unit away from the light source is modeled as:

attenuationFactor=1/(constAttenuation+linearAttenuation*D+quadricAttenuation*D^2)

Spot Light:

- A spot light is both directional and positional.
- It sends light rays in a specific directional region (using the attenuation data as well).
- If directional attenuation model does not affect light intensity, it can behave as a point light.
- As stated in previous OpenGL specifications that has fixed-pipeline lighting support (Ex: 2.0), “If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors.”

Extending Light Types

REng lighting system works independent of specific light parameters, allowing you to define your own light types which can be managed by the generic system. To extend the predefined lighting models:

1. Derive your own class from Light_Base
2. Fill in virtual methods as necessary and define your light parameters.
3. Define the public "static bool registerFunc();" method, and
    1. Add REGISTER_LIGHT_H(_LightType_) in your light header and REGISTER_LIGHT(_LightType_) in a cpp file.
4. Update RENG_LIGHT_TYPE_COUNT found in config.h to reflect the additional light types.
5. getTypeID must return a unique ID for each type, make sure it does. An light type ID cannot exceed (RENG_LIGHT_TYPE_COUNT-1).
6. define virtual abstract GPUShader methods (getShaderStructCode and synchWithPass).
    1. synchWithPass method should use mSynchLightIndex available in Light_Base.

In short: See how the pre-defined light types are defined.

Light Manager

Each light type defined completely is registered automatically to lighting manager, which is responsible for general management of scene lighting information.

- NOTE: The light object array for each light type can of maximum MaxNumOfLights_PerType size. Accessing a higher index in a shader will result in a shader compilation error. Also, OpenREng cannot assign more lights to a mesh, even if such lights can illuminate the object.
- NOTE: The lighting manager is not yet completed, and currently assumes that every light in the scene illuminates every object in the scene. No light culling or per-object light assignment is done currently, even though the following discussions may include such processes.

Lighting Passes

1. For each light in the scene
    1. Identify which objects can be lighted given that light
        1. If the light defines a bounding illumination volume, and if the mesh also defines such a volume, volume intersections are used to detect "colliding" light-mesh pairs.
        2. If a light can affect a single scene graph node hierarchy, the light-mesh assignments needs to be done only once.
        3. You cannot assign more than MAX_LIGHTS_PER_OBJECT lights per mesh.

- As a result, each mesh node is assigned to a list of lights (which is linked to a light node). The list may be empty, or in maximum, MAX_LIGHTS_PER_OBJECT size.
- Also, updating this information should be done in minimal time -> Do not try to update if unneccesary! TODO
- The memory overhead should also be minimal => Searching light-mesh pairs must be FAST. TODO

Rendering Meshes

1. On mesh render time, for each mesh
    1. Multiple objects can be illuminated by the same light sources
        1. I will address this issue later !!!
    2. For each of the light that affects this mesh
        1. Set light n'th uniform variables, by getting these values from the light object OR letting the light directly set the uniforms inside.
        2. ONLY the lights which match the given attenuation model will upload attenuation data! OR these lights can be skipped.

- Lights of the same type are put in the same array. The ordering of these lights is not defined.
- Q: How to support colors of integer type?
- Q: Precisions of members?

Lighting Demo

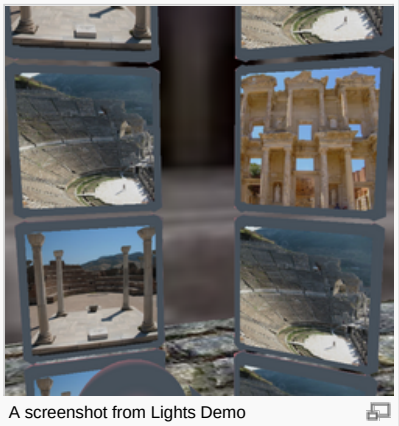
A demo is provided with the OpenREng library to showcase the basic use of the lights in the scene. Related materials are found in apps/bin/materials/lights folder, while some of the light-specific shaders are found in apps/bin/shaders/light folder. Additional screenshots can be found here

This demo supports the following variants of materials using a single light source using per-pixel or per-vertex lighting:

- Sun (Press 1 for per-vertex / Press 4 for per-pixel)
- Point (Press 2 for per-vertex / Press 5 for per-pixel)
- Spot (Press 3 for per-vertex / Press 6 for per-pixel)

Notes:

- The scene includes node animations which can be activated by pressing spacebar or 7.
- The default scene illumination can be activated by pressiong 0, which uses a material with higher ambient lighting values.
- This demo features images from the ancient city of Ephesus, Turkey.



A screenshot from Lights Demo

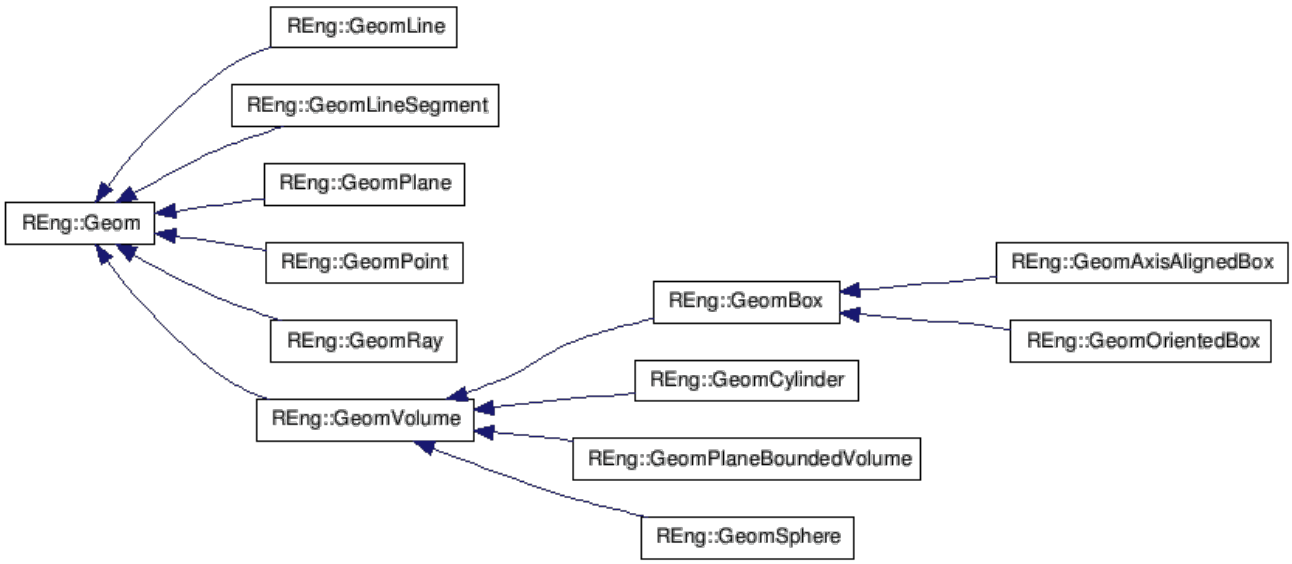
TODO

- Specify how object are linked to lights, which object is lit by which lights.
    - Specify an ordering of lights for each object. Shader dev may want to implement separate lighting equations for each light
- Implement a fast and effective synchronization procedure.

Geoms

Geometric entities that can be represented parametrically (not vertex-by-vertex) are represented as Geom objects. Related files in this component are Geom.h, GeomHelper.h and GeomRenderer.h. It does not depend on any other REng component and can be used separately if required (except the Renderer component).

Geom Class Hierarchy



This is the current hierarchy of geom classes. The intermediate (GeomBox and GeomVolume) is just virtual classes to hold common interfaces (i.e. they cannot be instantiated). More information can be found at source documentation.

Translation, Rotation and Scale of Geom Objects

Geom objects can be translated, rotated and scaled. These operations are similar to operations applied to nodes, yet there exists no parent-child relationship between geom objects.. Translation, rotation and scale operations are done in local space where the basis axis are parallel to the world-space.

- If a geom object is translated, then only *mPosition* attribute is altered accordingly.
- If a geom object is rotated, then since the operations are in local space the object is rotated around its center, rather than the world space origin. So, *mPosition* is not effected at all.
- Similarly, since scale is done in local space, it also does not effect *mPosition*.

GeomHelper

GeomHelper includes operations that compute the relationships between two geom objects which the geoms do not include. The currently considered relationships are:

- Getting distance between two geom objects.
- Getting squared distance between two geom objects (For performance reasons square variants of distances is provided.)
- Volume intersection tests
- Containment (Inclusion) test
- Merging two geom objects
- Side queries (Used between Plane - Any Geom)

Since the Geom includes a class hierarchy and many functions are overloaded, dynamic dispatch is required. This can have a negative impact on performance. Yet, the compiler can automatically detect correct overloaded function on compile time when object types are available.

GeomRenderer

This class is specially designed to receive geom objects and render them to screen as efficient as possible. It uses MeshGeomGenerator::getUnit*X* to generate unit geom objects of desired type *X*. Then, those unit geom objects are placed to correct position, orientation and scale by updating their model matrices. The final object is then rendered using GPUDrawer::drawMeshGeom.

In essence, this class supplies an abstraction for geom rendering purposes, by combining drawing and mesh generation tasks into a single method call.

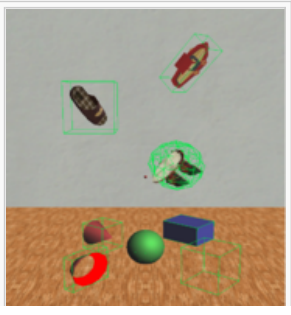
Tests

There are two tests for Geom component which are not completed: Unit test and stress test.

In addition, a visual approach to testing is adopted where an application is developed: Geom Objects Demo.

Geom Objects Demo

A visual application provides an easy way to test and observe bugs and serve as a basic example application for geom objects. This application can be found under *trunk/apps/GeomDemo* in source-code. There are several issues concerning geom primitives which are tested by *Geom Objects Demo* application.



- Several meshes are computationally generated representing geom primitives. Those are box, wire box, sphere, cone, and cylinder; for the time being.
- Several .3ds meshes are loaded with their textures.
- Both generated and loaded meshes are encapsulated in bounding boxes. There are 3 different choice of bounding boxes. Those are axis aligned box, oriented box, and sphere.
- Using those objects, several methods of GeomHelper class is tested, including distance calculations, intersections, and containment calculations.

- *Geom Objects Demo* is also used as the main test routine of Picking, which uses bounding boxes and intersection methods to achieve its objective.

A screenshot from Geom Objects Demo

Unit test - Absent

- It should be a test to check "whether it works correctly".
- How to identify the "ground truth" for complex operations (such as rotation operations) ?
- No separate unit test plan specification is required, as long as the code is written well (highly documented and structured).

Stress Test - Absent

- It should be a test that generates performance information about the implementation.
- The floating point performance on mobile platforms is expected to be low. It is a requirement to know what is the maximum number of operations that does not limit real-time behavior of the system.
- The stress test produces log files that can be easily compared.
  - TODO: Specify the log file structure. (Ex: Test: Intersection. Plane vs Plane. # of calls : 10000, time: 312ms, average time for 100 op's : 3.12ms, etc]
  - The outputs of mobile and PC device will be compared.
- *Test Type 1:* Perform the same operation on specific Geom object(s) a high number of times (ex: 1000).
- *Test Type 2:* Perform the same operation on randomly types Geom objects. This is used to prevent instruction caching and test dynamic dispatch performance.

Miscellaneous

Render Matrix Management

Introduction

A render matrix is a *basic* matrix that is used in the programmable shading pipeline to transform an input data (ex: point). The possible render matrices are:

- Model (specifies model transformation)
- View (specifies view transformation)
- Projection (specifies projection transformation)

The following variants are also possible and derived from basic matrices:

- Model*View, View*Projection, Model*View*Projection (1)
- Inverse, transpose and inverse of transpose variants of basic matrices and matrices noted above. (2)

Problem

Latest OpenGL versions (OpenGL ES 2.0 and OpenGL 3.1) does not define an interface to define regular render-based matrices, such as **model-view matrix,** *projection matrix'* and other matrices derived from those matrices. A module should be responsible to track/set render matrices and automatically manage their derivations. It is required for those internal matrices to be retrieved by shading programs using uniform variables.

Design

- RenderMatrixManager is the class that solves the problems above.
- Only model, view and projection matrices can be modified externally.
  - NOTE: (1) is automatically derived, (2) is updated when only needed.
- Special named uniforms are defined for shaders to automatically receive render matrix information. The auto-named uniforms start with "re_" prefix. (TODO: describe those and other possible auto-named uniforms in another section!)
- The following now-removed methods in previous OpenGL versions are provided withing !RenderMatrixManager:
  - glLoadMatrix: Updating matrix uniforms [Setters are not state-based as in previous OpenGL specifications.]
  - glPushMatrix / glPopMatrix : Using a non-limited dynamic size stack-implementation for storing matrix sequences, stacking operations are supported.

Frequently Asked Questions

Q: I have a problem?

1. RTM :) [I suppose your problem is about REng ? :)]
2. Join the mailing list, send an e-mail describing the situation.

Q: How can I check what OpenGL version my desktop driver supports?

- Download and run GLView (For Windows XP-Vista , MAC OS) BR
- Download and run GPU Caps Viewer (For Windows XP-Vista)

Q: My desktop drivers does not support OpenGL 3.0 or above? What can I do?

Don't worry yet, use the OpenGL ES simulators (From PowerVR or ATI). Below are some instructions for Visual Studio. Make sure that the ES SDK examples work in your machine (they should work if your desktop OpenGL version is 2.0 or 2.1), if not, then you can start to worry :)

Q: When building, there are errors like "Compiler cannot find cml/cml.h or il/il.h" etc. How to resolve this problem?

Option 1:

- Checkout -from svn- dependencies folder next to your REng folder.

- If you build a branch/tag folder source:
  - When configuring CMake files, make sure you UNCHECK the "BUILD_IN_TRUNK" option. This option is used to adjust the dependency folder, since your are now one folder down in hierarchy. Otherwise (you are building in trunk folder), leave that option ON.

Option 2:

- Manually find the required lib/include folders.
  - Can be done using CMake, but we need customizations for that. Left as an "exercise" to the user :)

Q: When building, there are errors like "Compiler cannot find a <function> for linking...

The compiler cannot find the lib (.lib in windows, .a or .so in unix) file in given lib paths.

- If you use dependencies folder structure, extract the libraries for your platform found in pre-comiled dependencies folder to the dependencies/lib folder.
- Note: Do not commit binary library files into lib directory!
- OR: You can build the dependencies (using the correct versions) on your platform and place your libs there. Most of the dependencies come with makefiles/project files from their original source. Use them.
- NOTE: You will need to install the shared binaries of the libs (if they are build as shared libraries)
  - In windows Place the corresponding dll's in precompiled packages into reng execution folders (ex: apps/bin).
  - In unix, you need to copy the .so files into /usr/local/ etc (I don't remember exact path, if you know it, fill it in please)

Q: I want the link the application to shared library build of REng? How can I do that?

1. Look at the CMakeLists.txt file in the application's folder. There should be some statement like: TARGET_LINK_LIBRARIES(MeshViewer REng_Static)
2. Change REng_Static with REng_Shared
3. Re-run CMake
4. "make" / the source tree build again (it will only trigger a linking command if you have not updated any other settings/files).

Q: The applications does not start, it crashes on start. What should I do?

Cause 1: Problem with OpenGL drivers:

- If you do not enable "Use OpenGL ES?" option when cmake generates makefiles, MAKE SURE YOUR GRAPHICS DRIVERS SUPPORT OPENGL 3.0.
- See above to find out your OpenGL Version.

Cause 2: Linking to incorrect libraries:

- If you used precompiled_vs8_sp1.zip or any other such file, make sure that you installed SERVICE PACK 1 for VS2005, that's what these libraries were compiled with, as stated in the name of the zip file.

Cause 3: You are using Visual Studio and running the projects from within VS:

- If you "debug/run" the project from VS, the "Working Directory" setting per project is important, yet it cannot be set by the CMake system details here .
  - You can manually set working directory: Project -> Properties -> Configuration Properties -> Debugging -> Working Directory -> Set to "$(TargetDir)", without quotes.
  - Note that if you erase *suo files / clean your build directory, this information will be lost and you have to manually set it again.

Cause 4: Some kind of internal REng bug:

- Check the log files, especially the rendersystem log!
- Zip and send all the log files generated to a REng developer.

Q: Even though a model is textured, it is shown as black. What is the problem?

Most probably, the texture could not be loaded into OpenGL. One cause may be that the texture uses indexed colors, which is not supported in OpenGL ES or 3.0. Check material system log file for any WARNings or ERRORs.

http://sourceforge.net/apps/mediawiki/openreng/index.php?title=FAQ

Category: Components

This page was last modified on 6 February 2010, at 13:44. This page has been accessed 66 times.

Powered By MediaWiki

Copyright © 2010 Geeknet, Inc. All rights reserved. Terms of Use