Cartel Documentation

Russell Gillette & Darcy Harrison

April 2014

1 Overview

Cartel is a mesh visualization and modification tool designed for simplicity and clarity. It is built on top of a basic OpenGL 3.3 environment using the GLFW windowing system, and enables complex rendering effects through the easy use of shaders and textures. Deployment and compilation are simple due to the use of few, and common, dependencies.

2 Dependencies

The libraries included with the build are for VisualStudio 2012 or later, and the source code includes some (though not many) C++ 11 operations, and thus will not work with earlier compilers. That said, there is nothing OS specific within the code, and alternative libraries are easily acquired at the below links.

2.1 GLEW

GLEW is the extension wrangler library for OpenGL that allows the use of newer OpenGL functionality on computers that do not support it. You will never have to do anything using this library, its just included, and thus worth mentioning. Its home webpage is here:

`http://glew.sourceforge.net/`

2.2 GLM

GLM is the open gl math extensions and provides a lot of useful functionality when interfacing with GLSL, the OpenGL shader language. This library provides types, and functions on those types used SOLELY FOR RENDERING. This is important, because the glm types do not offer double precision, and do not interface easily with Eigen. Use glm only for visual effects or when passing data to the GPU in main.

`http://glm.g-truc.net/0.9.5/index.html`

2.3 GLFW

GLFW is the windowing system used by Cartel. It is a common replacement for GLUT, and was chosen due to its nicer rendering loop and newer interface. This windowing library is cross-platform, allowing the deployment of this system across OSes, but does, at the time of writing this, make setting up this project on a linux system with integrated graphics difficult.

`http://www.glfw.org/`

2.4 Eigen

Eigen is the back-end linear-algebra library used by Cartel, and the largest source of complexity within the project. This library makes heavy use of templates, but is well documented. Documentation can found here:

`http://eigen.tuxfamily.org/index.php?title=Main_Page`

Please see §6 for common usage mistakes.

3 Classes

3.1 Mesh

Mesh is the main mesh container class, containing both the DrawMesh for rendering the loaded mesh, and the EditMesh for operating on the mesh. It exposes both draw operations from the DrawMesh, and edit operations from the EditMesh, so any functionality that has been added to either of those classes will need to be propagated into this class as well. Due to this separation of rendering and modification it is important to increment the edit count within EditMesh after each modification so that the changes can be properly propagated.

3.2 DrawMesh

The DrawMesh is responsible for mesh rendering, and contains the mesh data as a formatted block of memory with 32bit precision, rather than the 64bit precision offered when editing. This class is only ever populated from the EditMesh (not the other way). Data is stored within a VBuffer, which allows for various different data packing schemes, and uses the IBuffer to properly associate the data block with faces.

3.3 VBuffer

The VBuffer class contains the allocated block of memory with a 32bit precision version of the mesh. It also contains the packing information for the data, such as the offset to the start of a given attribute (position/normal/texture data) and the stride between specific elements.

3.4 IBuffer

The IBuffer class is a simple buffer containing only the face index (or other index) information to be used in rendering the mesh.

3.5 EditMesh

The EditMesh class contains a high precision version of the mesh in a half-edged data structure (see §5). It is from this mesh which the DrawMesh is populated.

As of right now, normals are not stored in the edit mesh, but are calculated upon transfer to the DrawMesh as face normals (thus vertices are duplicated per face). Texture Coordinates are not saved.

3.6 OBJLoader

This class does what you would expect it to. It parses .obj files. An interface from which to use this functionality is provided in "MeshUtils.h".

3.7 WorldState

This class is a singleton to contain "world" or global data, and is responsible for toggling between shaders, returning the current shader in the GL state, maintaining the tranformation matrices, and storing materials, textures, and lights to be used inside of shaders. THIS CLASS IS NOT RESPONSIBLE FOR INPUTS OR WINDOW INFORMATION. This global singleton is accessible in main as "w_state".

3.8 ControlState

This class is a singleton responsible for input and window state, as well as all GLFW callbacks. It is accessible in main as "c_state".

3.9 RenderState

RenderState objects contain the GLbuffer id's and VertexArray id's that have been allocated by opengl. When passed to the Mesh as initialization it specifies which set of buffers to use when rendering. It is safe to use only one RenderState, but performance may be gained by thoughtful allocation.

3.10 Texture

Textures are as of right now un-used, but the class is provided should texturing or texture effects be desired for rendering. This class is meant to be an interface for texture-type specific implementations, and thus should never be used on its own.

3.10.1 Texture2D

Defined in "TextureTypes.h", this class extends Texture and provides all of the necessary state setup for use with 2D Textures. (Note that these textures must be square. For non-square textures, Rectangle Textures will need to be implemented)

3.10.2 TextureCubeMap

Defined in "TextureTypes.h", this class extends Texture and provides all of the necessary state setup and interfaces for use with Cube Map Textures.

4 Auxiliary Functions

4.1 MeshUtils.h

"MeshUtils.h" contains utility functions that operate on meshes, or construct meshes, but which are not specific to a given mesh.

4.2 ShaderUtils.h

"ShaderUtils.h" provides utility functions for shaders, such as functions to load shaders from files, build shaders, and compile shader programs.

4.3 TextureUtils.h

"TextureUtils.h" provides utility functions for textures including the construction of texture objects from files.

5 Half-Edge Data Structure

Cartel uses a "half-edge" data structure. Each edge of a 2-manifold triangle mesh is represented by a pair of directed "half-edges", one on each adjacent face or hole. The half-edges form a counter-clockwise directed loop by linking to the next (forward adjacent) half-edge in the face or hole. The triangles of the mesh are each defined implicitly by the directed loop of 3 half-edges. A half-edge also references the *vert*ex it originates on, the *face* it resides in, and the *twin* half-edge which is resides on the opposite side of the edge.

5.1 Components

Each half-edge requires:

1. The face it belongs to
2. The next half-edge in the face loop, in a counter-clockwise direction. These two half-edges share exactly one vertex.
3. The twin half-edge that is adjacent on the same vertices but in a different face. It is directed opposite of this one (ie. If this half-edge is directed $v_0 \rightarrow v_1$ the twin is directed $v_1 \rightarrow v_0$.
4. The origin vertex of this half-edge. A half-edge directed $v_0 \rightarrow v_1$ will store a reference to v_0. The next half-edge in the loop

These four components are sufficient to implement all manner of interesting geometry processing algorithms. Given an arbitrary half-edge h in a triangle, the vertices can be retrieved via:

$$\begin{aligned} v_0 &= h.vert \\ v_1 &= h.next.vert \\ v_2 &= h.next.next.vert \end{aligned} \tag{1}$$

In order for the mesh to track the all the triangular faces, it is sufficient to store a reference to an arbitrary half-edge in each face. The vertices can be easily retrieved via the directed list of half-edges as shown in Equation (1).

For easy traversal of a vertex 1-ring, each vertex stores a reference to an arbitrary half-edge originating from it.

5.2 Boundaries

Meshes are often not closed ie. there are some edges which do not have adjacent triangles on (at most) one side. The polygonal region of empty space is referred to as a *hole*, edges adjacent to it are referred to as boundary edges. Our half-edge data structure handles places half-edges within the hole polygon so that every half-edge has a well defined twin. An alternative option (not pursued in this library) is to use a sentinel value to indicate when a half-edge does not have a twin. There are other benefits to our approach which merit its use. Half-edges in holes reference a special face reserved for holes. Holes are essentially treated just like any other face, except they have a special index, can have arbitrary degree (ie. not restricted to triangles), and are wound in opposite direction (ie. clockwise).

5.3 Common Operations

5.3.1 Face → Vertex Iteration

Given a face f_i, we can visit all the vertices in the face by starting from the arbitrary half-edge $H(f_i)$ associated with it.

$$\begin{aligned} h &= H(f_i) \\ v_0 &= h.vert \\ v_1 &= h.next.vert \\ v_2 &= h.next.next.vert \end{aligned} \tag{2}$$

5.3.2 Vertex 1-Ring Iteration

Given a vertex v_i, we can visit all of the vertices $\{v_j\}$ which share an edge with v_i by starting from the arbitrary half-edge $H(v_i)$ associated with the vertex, then iterating around the edges in a clockwise manner.

$$\begin{aligned} h_0 &= H(v_i).twin \\ v_0 &= h_0.vert \\ \\ h_j &= h_{j-1}.next.twin \\ v_j &= h_j.vert \qquad j = 1, 2, \ldots, \mathbf{Degree}(v_i) - 1 \end{aligned} \tag{3}$$

5.3.3 Vertex → Face Iteration

Given a vertex v_i, we can visit all of the faces that are adjacent on this vertex by starting from the arbitrary half-edge $H(v_i)$ associated with the vertex, then iterating around the faces in a clockwise manner.

$$\begin{aligned} h_0 &= H(v_i) \\ f_0 &= h_0.face \\ \\ h_j &= h_{j-1}.twin.next \\ f_j &= h_j.face \qquad j = 1, 2, \ldots, \mathbf{Degree}(v_i) - 1 \end{aligned} \tag{4}$$

5.3.4 Face Neighbor Iteration

Given a face f_i, we can visit the three neighboring faces by visiting the half-edges of f_i and their twins.

$$\begin{aligned} h &= H(f_i) \\ f_0 &= h.twin.face \\ f_1 &= h.next.twin.face \\ f_2 &= h.next.next.twin.face \end{aligned} \tag{5}$$

5.4 Implementation Details

The half-edge data structure is implemented more or less as described in § 5.1. All arrays are implemented via *std::vector*$< T >$.

- There is an array of *half_edge* structures each of which stores: the integer index of the *next* half-edge in its directed face loop, the integer index of the *twin* half-edge adjacent on the same vertices, the integer index of the *face* this half-edge is associated with, and the integer index of the *vert*ex that this half-edge originates out of.

- There is an array with an entry for each vertex, which stores the integer index of an arbitrary half-edge which originates from that vertex. There is another array of *Eigen::Vector3d* objects for the location of each vertex in space.
- There is an array with an entry for each face, which stores the integer index of an arbitrary half-edge which defines that face.

5.5 Rationale

Storing indexes can be problematic when deleting half-edges, faces, etc. Our current approach switches the index at the back side of the id array for the position being deleted prior to removing the index, such that no space is left in the array. This approach has the unfortunate drawback that it re-orders ids, and thus when traversing one of the index lists it is important to iterate in reverse order (such that all out of order ids have already been traversed).

In our implementation we considered many alternatives to this scheme, but found all of them to have their own problems. These alternatives include: Storing permanent ids, flagging objects as deleted but not removing them, or else storing ids within a linked list data-structure (each element allocated on the heap). Permanent ids increases the lookup time, requiring either a hash-table or binary look-up. Hash-tables incur a non-trivial overhead, and binary look-up increases the traversal time of the mesh significantly. Flagging objects as deleted results in non-existent elements remaining, and thus requires the non-intuitive check for deleted elements while traversing the mesh. Finally, linked-list data stores do not allow random access, thus having a large traversal time (unless pointers are stored). More importantly, each element would require an allocation, which is extremely costly.

6 Common Mistakes

6.1 Eigen Mistakes

- std data-structures must use the aligned allocators provided by eigen

  ```
  std::vector<Eigen::Vector3d, Eigen::aligned_allocator<Eigen::Vector3d> >
  ```

- due to implicit type casts, sometimes operations will not work. This is resolved by proper placement of parenthesis

  ```
  <double> * (Eigen::Rotation2Dd(<double>) * <Eigen::Vector2d>)
  ```