

Computer Graphics 2: Term Project

Catmull-Clark Subdivision on GPU

Using Compute Shaders

Abhishek Sachdev #7851876
5-1-2021

Contents

Motivation.....	2
Catmull-Clark Subdivision	2
Subdivision Rules	2
Implementing Catmull-Clark Subdivision on CPU	4
Performance Optimizations	4
Compute Shaders.....	5
Implementing Catmull-Clark Subdivision on GPU.....	5
Input/Output.....	5
Compute Shader Invocations.....	6
Results.....	6
Comparing performance data.....	7
Addressing the Bottlenecks and Future Work	7
Extra feature	8
References	9

Motivation

In Computer Graphics, a subdivision surface is a curved surface represented by a coarser polygon mesh and some subdivision rules. Subdivision rules describe an operation that can be applied to the coarser polygon mesh to refine it and produce a smoother surface representing the underlying curved surface. Catmull-Clark subdivision surfaces are one of the popular subdivision surfaces. The purpose of this project is to understand how the Catmull-Clark subdivision works and perform optimizations on this algorithm by using Compute Shaders to perform the calculations on the GPU.

Catmull-Clark Subdivision

Catmull-Clark subdivision algorithm was derived by Edwin Catmull and Jim Clark in 1978. The subdivision rules work on meshes of arbitrary manifold topology. The algorithm was obtained from generalizing the B-spline surface patch representation; thus, the resulting subdivided surface approximates the underlying bi-cubic uniform B-spline surfaces for the topology. This subdivision is a quad-based subdivision scheme and the resulting polygon mesh after the subdivision steps consists of quads.

Subdivision Rules

The rules of the subdivision are described as follows:

1. Face points are obtained by calculating the average of all vertices defining a face.

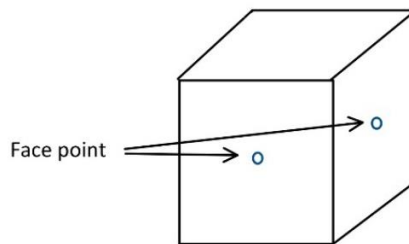


Figure 1: Face points.

2. Edge points are obtained by calculating the average of the midpoint of the old edge with the average of the previously calculated face points of the faces that share that edge.

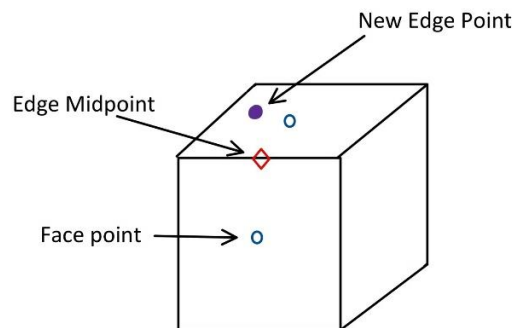


Figure 2: Calculating New Edge points.

3. New vertex points are obtained through the following equation:

$$v = \frac{Q}{n} + \frac{2R}{n} + \frac{S(n-3)}{n}$$

Where, Q is the average of the face points of all faces connected to the old vertex. R is the average of midpoints of all edges connected to the old vertex, S is the old vertex point, and n is the number of connected faces for the vertex.

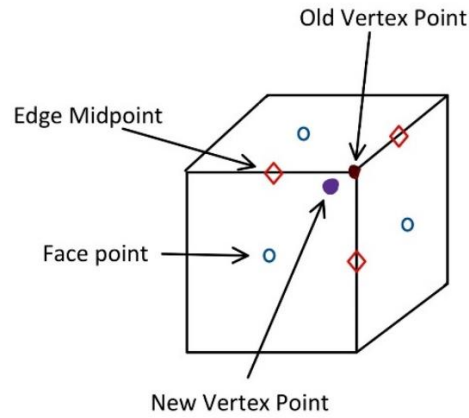


Figure 3: Calculating New Vertex points.

After computing the points, the new mesh is formed by calculating the following edges:

- Connecting each face point to the edge points of the edges defining the old face.
- Connecting each new vertex point to the edge points of all old edges connected to the vertex point.

Once the edges are calculated new faces are defined as those enclosed by the new edges. The figure below shows the result of applying the subdivision algorithm to a cube, applying 5 levels of subdivisions results in a cube approximating a sphere.

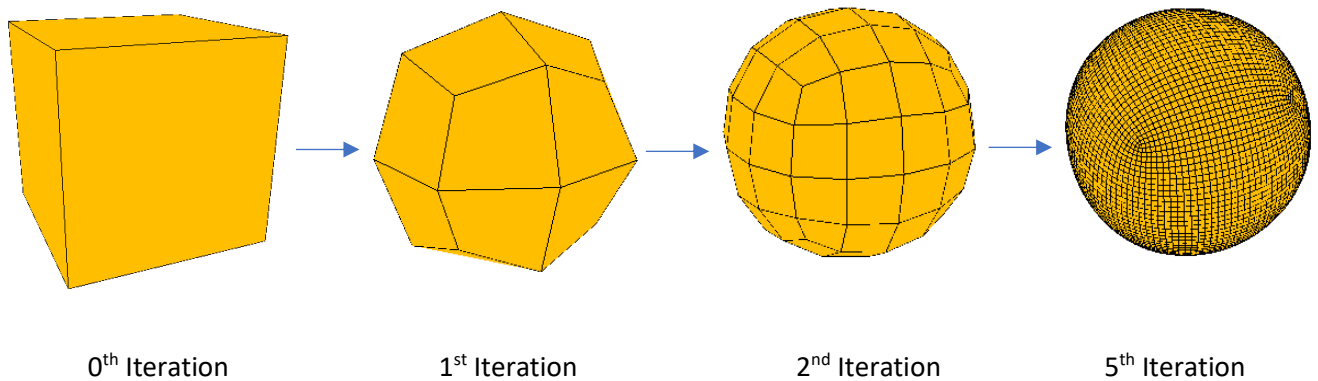


Figure 4: Applying Recursive Catmull-Clark Subdivision on a Cube

The vertices that are not connected to exactly four faces in the original geometry are defined as extraordinary points because the resulting subdivision at these points does not produce a B-spline surface and the surface at those points is not C^2 continuous i.e., the curvature between the faces connected by these vertices is not maintained. These vertices can be seen in Figure 4 as the eight original vertices defining a cube. From the figure above it can be noted that these vertices produce sharp corners and result in a geometry that is C^2 continuous on normal points and C^1 continuous on extraordinary points.

Implementing Catmull-Clark Subdivision on CPU

First, to understand how the algorithm was implemented on GPU, we need to look at its CPU implementation as this implementation was translated to work on GPU with some modifications. This implementation is written in C++ using OpenGL. The program starts by reading a .obj file and creates a new Mesh object. For CPU implementation this mesh contains a vector storing Vertex objects and a vector storing Face objects. A Vertex object contains an id, this id defines the position of the Vertex in a mesh. For example, if the Vertex is stored at the first position in the mesh it has id 1. Each Vertex stores references to its connected Faces and other Vertex objects which together form an Edge.

A Face object contains an id, which like Vertex defines where the location of Face in Mesh. Face stores the reference to its 4 vertices that enclose it. The Edge struct stores the reference to two vertices and the edge point. The implementation of the algorithm can be found in subdivision.cpp file in the Mesh* cc_subdivide(Mesh*, GLuint*) method. The method starts the subdivision by creating vertices for the subdivided mesh from the face points of the Faces in the mesh. Next, all the edge points are calculated by looping through individual edges of a Face. These edge points are stored as new vertices for the subdivided mesh. Lastly, the program loops through all the Faces in the Mesh; calculates new vertex points for vertices connected by the edges and creates new Face objects for the subdivided mesh. This last step can be refactored as looping through all the old vertices first to calculate new vertices and then creating a new mesh by looping through the faces of the old mesh.

Performance Optimizations

In the implementation described above, the program loops through the faces to calculate the face points, then loops through the edges to calculate the edges and finally loops through the vertices to calculate the new vertex positions. These loops can be removed by using GPU to perform the individual calculation of points in parallel. Further, the calculation of edge points and new vertices do not rely on each other and they do not need to be performed sequentially. This means that they can be put on GPU, calculated in parallel and the results can be aggregated at the end to create the new subdivided mesh. The last step of the subdivision where the program creates a new mesh can be pushed to GPU as well to reduce the load on the CPU and avoid memory copies. Compute Shaders can be used here as they provide the capability to define the number of threads to be executed in parallel thus providing more control on the parallelization of the program.

Compute Shaders

A Compute Shader is a Shader Stage that is used for computing arbitrary information. It is generally used for tasks not directly related to drawing on the screen but performing CPU-intensive calculations in parallel. The number of executions of the shader is defined by the function used to invoke the shader. There are no user-defined inputs and outputs for the shader in the execution model. Thus, an input for the shader must be provided by fetching data from the appropriate buffer interfaces, similarly, to obtain the results of the computations the data must be explicitly written to the buffers. Compute shaders work on a concept of work group which is the smallest amount of compute operations that the shader can execute. This is defined on the invoke call by the user. The space of these groups is three-dimensional, so a shader has X, Y, and Z dimensional groups. Setting Y and Z dimensions to 1 will result in a one-dimensional compute shader invocation which can perform calculations on a one-dimensional array. This provides a useful abstraction to perform 1D, 2D, or 3D calculations in parallel. The work groups may contain multiple invocations of the shader also known as the local size of a group; think of multiple threads within a single thread group. The local size of the work group can be set by the user and the optimal size is derived by comparing performance times of local sizes for a particular system. For this project, the local size of each work group is set to (1,1,1).

The Shader has access to a few input variables where `gl_GlobalInvocationID` is used extensively in the project.

$$\text{gl_GlobalInvocationID} = \text{gl_WorkGroupID} * \text{gl_WorkGroupSize} + \text{glLocalInvocationID}$$

Thus, in case of (N, 1, 1) work group size and local size (1,1,1), N total invocations of the shader are launched, and each invocation has `gl_GlobalInvocationID` ranging from (1,1,1) to (N,1,1). This abstraction provided by the shader can be used to index into a one-dimensional array of size N and perform operations for a particular index by: `array[gl_GlobalInvocationID.x - 1]`.

Implementing Catmull-Clark Subdivision on GPU

Input/Output

To implement the Catmull-Clark subdivision on GPU, input and output for the Compute Shader need to be defined. The program uses Shader Storage Buffer objects which are read-write storage buffers as the input and output buffers for the shader. The CPU program will need to write the appropriate data to these buffers to be used by the shader. The shaders can only work with primitive data types and structs. Thus, the data structures defining a mesh were changed from the CPU implementation to remove the use of classes and pointers. The new implementation stores the geometry in GPUFace, GPUEdge, and GPUVertex structs. GPUFace instead of storing references like the CPU implementation, stores the ids of its vertices in an int array, and similarly GPUVertex stores ids of its connected faces and edges in int arrays. GPUEdge class contains the ids to two vertices that make up an edge and the id of face it belongs to. The positions are stored as struct Vertex containing float x, y, and z position values instead of `glm::vec3` as copying structs containing glm types to the shader was causing failures on my device. GPUMesh object on the CPU stores the following structs for rendering the geometry on the screen. After the subdivision is completed the GPU writes back GPUFace and GPUVertex back to the CPU.

There are 5 total Shader Storage buffers for use by the Shader. 3 of which are used to provide shader with input GPUVertex, GPUEdge, and GPUFace structs. The remaining 2 buffers are the output buffers where the shader writes the output GPUVertex and GPUFace for the subdivided geometry. Note that the GPUEdge structs are formed on the CPU for use by the shader.

```
struct Vertex {
    float x, y, z;
};

struct GPUVertex
{
    int id;
    int currentFace;
    int currentEdge;
    int faces[10]; // max 10 connected faces
    int edges[20]; // max 20 connected edges
    Vertex v;
};

struct GPUFace
{
    int id;
    int vertices[4];
    Vertex facepoint;
};

struct GPUEdge {
    int id;
    int face;
    int vertices[2];
};
```

Figure 5: Structs used by GPU Implementation.

Compute Shader Invocations

After writing the input data to the buffers are binding them to the appropriate location. The compute shader dispatch calls are made 4 times in total. In the first call, the compute shader calculates the face points for every face in the geometry by invoking the compute shader to run for each face by using `glDispatchShader(faces.size, 1, 1)`. Face points are needed by the other dispatch calls thus `glMemoryBarrier()` is called for the input face buffer to make sure writing face points to the buffer is finished before any further calls are made that read the face points to avoid race conditions.

The second and third dispatch calls are made to calculate new edge points and vertex points. Note that `glMemoryBarrier` is not called between the invocations here as they can be executed in the parallel as no state is shared between the calculation of edge points and new vertex points. This means that there are `edges.size + vertices.size` invocations of the shader running in parallel. Finally, to aggregate the results before constructing the geometry `glMemoryBarrier` call is made to finish all the above calculations. The last invocation of the shader creates new subdivided geometry and writes it to the output buffers. The last invocation is a single invocation instead of individual invocations for each face as the state among the invocations is shared resulting in race conditions. This can be solved by implementing a locking mechanism for buffer writes on the GPU, however, that was beyond the scope of the project.

Results

The GPU program produces optimal and correct subdivisions like the CPU counterpart. The provided implementations contain the subdivision code for the GPU and CPU implementation, the default is GPU which can be changed to use CPU by removing the `#define GPU_MODE` statement in `render.cpp` file.

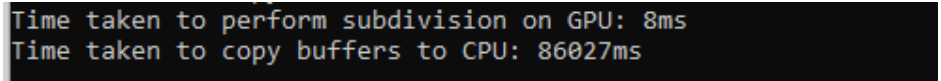
Comparing performance data

For the 5th iteration of the cube subdivision: CPU implementation takes 6.39s and the GPU implementation takes 5.58s.

For the 6th iteration of the cube subdivision: CPU implementation takes 99.21s and the GPU implementation takes 86.01s in total.

The performance times were measured on a device with RTX 2060 graphics card, 16GB ram, and Windows 10 operating system.

From the data above it can be noted that the GPU implementation has around a 10% reduction in the time taken to calculate the subdivisions. This performance difference is not that significant because of a major performance bottleneck in the implementation. This performance bottleneck is the result of glMapBuffer calls which maps the data stored on a buffer object in the client's address space. These calls are made to copy the data back from the output buffers written by the GPU to the CPU for rendering the geometry. The copy operation took most of the time in the 6th iteration of the cube subdivision while the GPU invocations took only about 8ms, as shown in Figure 6.



```
Time taken to perform subdivision on GPU: 8ms
Time taken to copy buffers to CPU: 86027ms
```

Figure 6: Performance time for GPU Implementation.

Addressing the Bottlenecks and Future Work

As mentioned above the data being copied back to the CPU for rendering is the major performance bottleneck in the implementation. Addressing this issue is straightforward but requires more development time and changes to the current implementation. To resolve this issue, the rendering pipeline can be changed to read the vertices directly from the output shader storage buffers filled by the compute shaders instead of copying the data back to the CPU to fill VBOs. This will require changes to the vertex shader to read the correct vertices. Compute shaders do not need to be many changes in this case, for instance, in the next iteration of a subdivision, data from the previous output buffers can be copied to the input buffers instead of the CPU roundtrip.

Extra feature

The program also comes with obj loader support that works for quad mesh geometries and can read vertices and faces for a geometry. The figures below show the some outputs from the program.

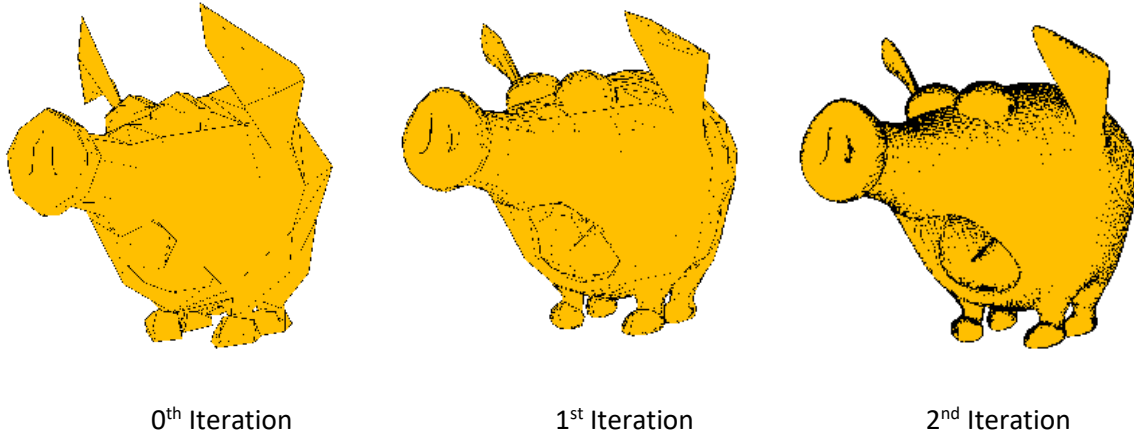


Figure 7: Catmull-Clark Subdivision on pig.obj by Keenan Crane

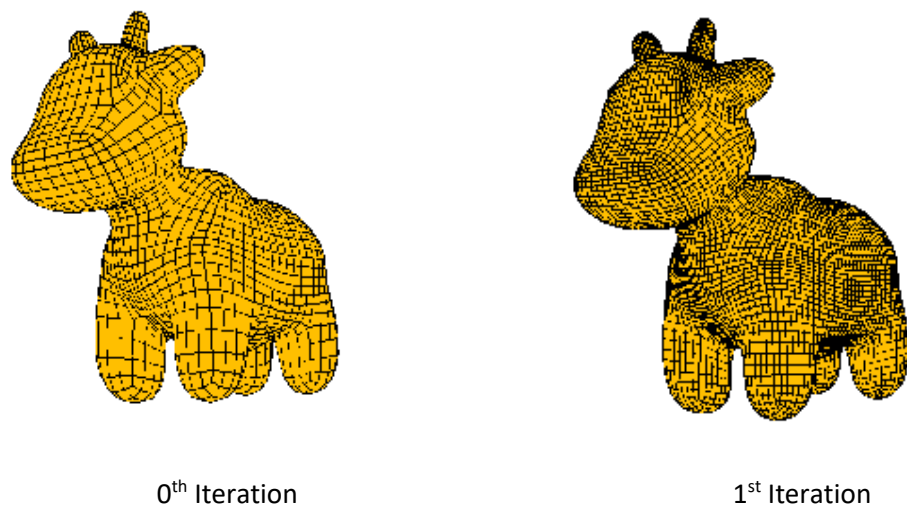


Figure 8: Catmull-Clark Subdivision on spot.obj by Keenan Crane

References

Catmull, E., and J. Clark. 1978. "Recursively Generated B-Spline Surfaces on Arbitrary Topology Meshes." *Computer Aided Design* 10(6), pp. 350–355.

https://www.khronos.org/opengl/wiki/Compute_Shader

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glMapBuffer.xhtml>

https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

Obj files by Keenan Crane: <https://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository/>