Triangle Strip Mesh Construction from Indexed Triangle Meshes using Greedy Algorithm

I Gede Govindabhakta 13519139
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13519139@std.stei.itb.ac.id

***Abstract*—Any complex shape rendered using Computer Graphics is a composition of primitive shapes such as points, lines, quads, and most commonly triangles. Among the factors that contribute to the performance of rendering is the amount of data transferred from the CPU to the GPU, in the case of this paper, the number of vertices used to represent a mesh. Triangle strips are a method of representing a mesh reduce the number of vertices stored to render an object. However, most formats for saving a mesh's data are in the form of indexed meshes. This paper discusses a greedy algorithm approach for the generation of triangle strip representations of indexed triangle meshes**

Keywords—Computer Graphics, Triangle Strip, Mesh, Greedy Algorithm

I. Introduction

In computer graphics, the geometry and topology of an object is represented by a mesh, consisting of vertices (composed of the corresponding x, y, and optionally z values). Rendering this object requires defining faces, requiring a varying number of vertices depending on the type of primitive used (1 for points, 2 for lines, 3 for triangles, and 4 for quads). Most used for both 2D and 3D rendering is the triangle primitive because it is the simplest 2-dimensional primitive which can also represent any other 2D primitive through composition.

In the case of triangles, rendering a face will require 3 vertices. For a 3D object such as a cube, a total of 12 faces are required (2 triangles for each square face of a cube). It is apparent from this example that many of the 36 vertices required are duplicates representing the same point. To handle this, graphics APIs such as OpenGL provide methods of reusing vertices by representing faces as pointers to a specified vertex in an index buffer. This reduces the number of vertices sent by removing the duplicates and representing faces with 3 indexes, each representing a vertex.

To further reduce the amount of data sent, we can reduce the number of indices by rendering meshes as triangle fans and strips. Relevant to this paper, triangle strips represent meshes as a sequence of triangles adjacent to each other and represents them as a sequence of vertices alternating between upright and upside-down triangles.

Most 3D modelling software and 3D model file formats such as OBJ store mesh data as indexed triangles, therefore this paper will discuss a greedy approach for converting indexed triangle meshes into ready to use triangle fan meshes.

II. Theory

A. Triangle Meshes

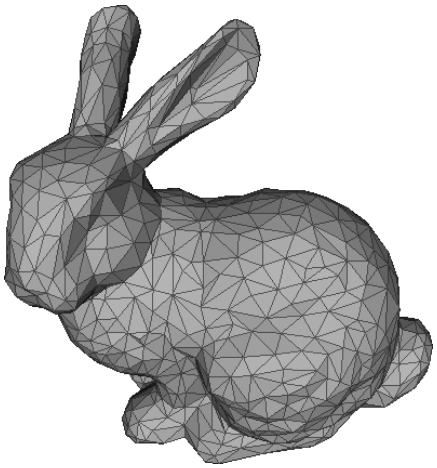

Figure 1: Example of a triangulated mesh [4]

Models are typically composed of a group of points in space called vertices which are used to compose triangles called faces. Each vertex is a vector, normally with 3 components (x, y, and z). Each face is a group of three vertices. In a non-indexed triangle mesh, the faces are stored as-is, keeping the vertex data for each point in the triangle, regardless of whether the vertex is shared with other triangles. This causes duplicates of vertices to be a common occurrence. The total amount of vertices needed to store n number of triangles is $3n$.

Non-indexed triangle meshes are often the starting point when learning computer graphics. They are as simple and show the inner workings and behavior of shaders. However, they are hard to maintain, read, and edit even for small objects with few faces. They are quickly abandoned for the more practical (and compact) indexed triangle meshes.

B. *Indexed Triangle Meshes*

Indexed triangle meshes solve the issue of duplicate vertices by storing the information required to build faces separate from the vertex information. The faces contain pointers or "indexes" to the corresponding vertex, thus eliminating any duplicate vertex data. For example, representing a 2-dimensional square with an indexed triangle mesh with vertices [(1, 1), (1, 0), (0, 0), (0, 1)] will require indices [(0, 1, 2), (0, 2, 3)].

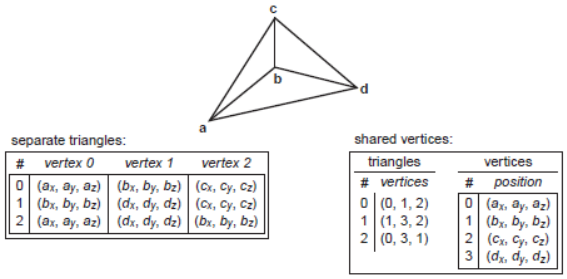

Figure 2: Example of shared vertices in an indexed triangle mesh [1]

The indexes normally refer to the order of the vertexes stored sequentially. The starting point for the vertices for graphics APIs normally starts at 0, however model file formats may deviate from this such as OBJ that start vertex indexing from 1.

The total amount of vertices needed to store *n* number of triangles is the number of unique vertices. There is additional data stored, the indices (which are usually unsigned integers, a third of the size of a vertex which is represented as a 3-dimensional vector), which is 3*n* in size, with n being the number of triangles (faces).

Indexed triangle meshes are the most common used way of representing mesh data. Most 3d file formats can be easily interpreted into ready to use indexed triangle mesh data. Graphics APIs also can render this data with very little modification from rendering a non-indexed triangle mesh.

C. *Triangle Fan Meshes*

Triangle fan meshes are the less applicable cousin of triangle strip meshes. Much like the name implies, they are rendered to the shape of a fan, with all faces sharing a common vertex. This way of representing a mesh is very compact but not applicable (to a practical extent) to a wide variety of meshes. The number of vertices stored is the same as indexed triangle meshes, however the number of indices stored is reduced to match the number of vertices stored, making it very compact.

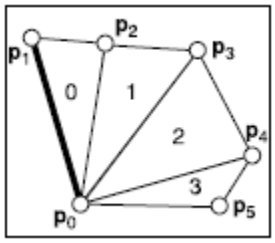

Figure 3: A triangle fan [1]

D. *Triangle Strip Meshes*

Triangle strip meshes represent meshes as a sequence of adjacent alternating upright and upside-down triangles, completely opposite to triangle fans which are essentially sequences of upright triangles (relative to their neighbors, thus the common vertex). For example, representing the same 2-dimensional square in the indexed triangle mesh example with a triangle strip can be done with the sequence [1, 2, 0, 3], using 2 less indices than the previous example.

Both triangle strips and fans have limitations when representing a single mesh, therefore multiple strips or fans may be required to represent a single object.

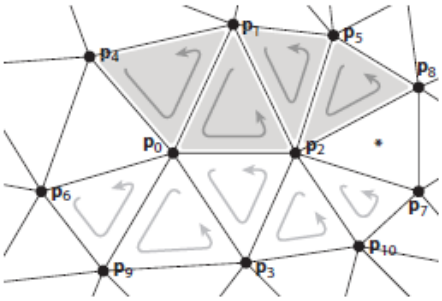

Figure 4: Two triangle strips in context of a larger mesh. Note that neither strip can be extended to include the triangle marked with an asterisk [1].

The total amount of indices required by a triangle strip to represent *n* number of triangles is *2+n*. Visually, this can be interpreted as first defining 1 line using 2 points then adding alternating points to form the triangle sequence *n* times. The alternating top-bottom pattern for the vertices must preserve the order (clockwise or counter-clockwise) of the faces defined which is normally used to calculate the surface normal of a face for uses such as lighting.

The compactness of triangle fans has diminishing returns the longer the strip gets. Because of this, having multiple shorter strips can be more beneficial than having long strips with remaining artifacts of leftover triangles which were unable to be added to the strip.

strip length	1	2	3	4	5	6	7	8	16	100	∞
relative size	1.00	0.67	0.56	0.50	0.47	0.44	0.43	0.42	0.38	0.34	0.33

Figure 5: Diminishing returns on longer strips [1]

Several programs have been created before the writing of this paper, notably the FTSG (Fast Triangle Strip Generator) by Xinyu Xiang and collaborators [2] (1999) and Oliver Matias Van Kaick and collaborators [3] (2004), among others. Both of which are very efficient and effective in the generation of triangle strip meshes.

E. Rendering Triangle Strips in OpenGL

OpenGL or Open Graphics Library is an application programming interface (API) for rendering 2D and 3D graphics. It is cross-platform and has bindings for many languages, though commonly used with C++. Applications that use OpenGL vary from video games to data visualization. However, as a low-level graphics API, it is often abstracted by other programs such as game engines.

Rendering a mesh in OpenGL typically involves storing the vertex data in a vertex buffer in the CPU, then sending it to the GPU. Using an indexed mesh, additional data is stored in an index buffer, which is also sent to the GPU. Both buffers are then passed into programs that run in the GPU called shaders such as the vertex shader and fragment shader.

OpenGL works as a state machine, thus inserting values into a buffer requires that buffer to first be selected or to "bind" it, then passing in the values. On render time, we can determine the number of vertices to render and the way we want to render it. A quick (oversimplified) example of how a mesh may be rendered in an OpenGL program is presented below.

```
// creating shader program and buffers,
// creating window context
...

// bind and store data to a vertex buffer
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBufferData(GL_ARRAY_BUFFER, 4, vertices, STATIC_DRAW);

// bind and store data to a index buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 1);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6, indices, GL_STATIC_DRAW);
...

// render loop
while(something)
{
     // Draw 2 triangles, indices are integers, no offset
     glDrawElements(GL_TRIANGLES,  2, GL_UNSIGNED_INT, ...);
}
...
```

Figure 6: Simplified example of how OpenGL renders an indexed triangle mesh

Note from the example that the mesh only has 1 index buffer, how would it handle a mesh that cannot store all it's indices in a single buffer, for example in a triangle strip where the mesh cannot be represented in a single strip? We can do this by sending the index buffers for each strip during run time in a for loop. This requires some modification of the render loop show below.

```
...
// render loop
while(something)
{
     foreach(int[] strip: TriangleStrips)
     {
          glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 1);
          glBufferData(... , strip, ...);
          glDrawElements(GL_TRIANGLE_STRIP, ...);
     }
}
...
```

Figure 7: Simplified example of how an OpenGL program renders an indexed triangle strip mesh

The purpose of the usage of triangle strips is to reduce the total amount of data sent through both the index and vertex buffer, thus increasing performance during rendering.

F. Greedy Algorithm

A greedy algorithm is any algorithm that at each decision-making step, selects the best option based on selection function at the local or current scope, with hopes of reaching a global optimum. An example would be to buy the most value out of a purchase of multiple items by continuously choosing the cheapest item until no money is left available.

Elements of a greedy algorithm are:

- A candidate set, consisting of possible elements of a solution
- A solution set, consisting of elements of the candidate set chosen for the solution
- A selection function, which decides the best option at a given step.
- A feasibility function, which decides whether a candidate can be selected
- An objective function, which returns a value for a solution.

The elements are used by a greedy algorithm as follows:

- The selection function is used to select the best current available option
- The feasibility function is used to determine if the selected option is feasible and adds it to the solution if true

- The selected option is removed from the candidate set
- The process is repeated until no candidates remain or a solution is found

III. Implementation

The greedy algorithm implemented is a *greedy by length*, based on how longer strips are more compact where compactness is defined as the number of indices required for a triangle strip over an indexed triangle mesh ($n+2/3n$, n is the number of triangles). At each step, it tries to insert a face into a compatible strip. If two strips or more are available, the one with the longest length is given the new triangle.

A. Candidate Set

The candidate set for this algorithm is the set of available triangles (faces) given. Each triangle has indices that point to a vertex. The vertex data themselves however are not part of the candidate set, as they are additional information relevant to the rendering of the mesh and not the construction of the triangle strip.

B. Solution Set

The solution set for this algorithm is the set of sequences representing triangle strip meshes. A solution is valid if each triangle is included in only one strip. In other words, no strips overlap with each other.

C. Selection Function

The selection function for this algorithm adds the newest face to an available compatible strip. If multiple strips are compatible, the algorithm chooses the longest of the group. The intent of this decision making is to create the lon

D. Feasibility Function

The feasibility function for this algorithm determines whether a triangle can be added to a strip. A triangle can be added to a strip if for every even addition to a strip (the first triangle is counted as 0, not 1) when represented as a circular list, there exists 2 vertices which match the last 2 vertices in the sequence list in reverse order. For example, the triangle [2, 5, 1] can be added to the strip [4, 0, 1, 2] because in reverse order and shifted by 1, the triangle is [1, 2, 5] which matches the final two vertices of the strip ([1, 2]). This is implemented because the vertex data used for this algorithm are uniform in winding (clockwise).

E. Objective Function

The objective function of this algorithm calculates the number of total indices used and divides them by 3*n*, *n* being the number of faces. Lower values represent better strips (more compact). The maximum value achievable is 0.33, which is determined by the limit to infinity of the compactness function for a single strip ($n+2/3n$)

F. Implementation

The following code is written in Javascript and describes the large picture of the steps taken by the algorithm.

```
function toTriangleFan(indices)
{
        let strips = [];
        let selected;

     for(let i = 0; i < indices.length; i++)
        {
            selected = {
                face: -1,
                length: -1,
                last: -1
            };

            for(let j = 0; j < strips.length; j++)
            {
                selected = select(strips[j], indices[i], selected,
j);
            }

            if (selected.length < 0)
            {
                console.log("found new", indices[i]);
                strips.push(indices[i]);
            } else {
                console.log("added", indices[i], " to ",
selected.face);
                strips[selected.face].push(selected.last);
            }
        }

        return strips;
}
```

Figure 8: Main algorithm for triangle strip generation

The selection function chooses the current compared strip if the last selected has a shorter length as follows.

```
// Selects the best strip to add to
function select(strip, face, selected, i)
{
        let c = compatible(strip, face);
        if (!c.status)
        {
            return selected;
        } else {
            if (strip.length > selected.length)
```

```
            {
                return { face: i, length: strip.length, last: c.last }
            } else {
                return selected;
            }
        }
}
```

Figure 9: Selection function for triangle strip generation

IV. Testing and Analysis

Testing is done with the following sample data containing a multidimensional array of the faces available in an unspecified mesh. The indices are a modification of the presented mesh presented in figure 2.

Indices		
6	0	4
4	0	1
6	9	0
0	2	1
1	2	5
5	2	8
2	7	8
0	9	3
3	2	0
3	10	2
2	10	7

Table 1: Faces used for testing

The algorithm produced the following results:

Strip	Sequence
1	6, 0, 4, 1
2	6, 9, 0, 3, 2, 10, 7
3	0, 2, 1, 5
4	5, 2, 8, 7

Table 2: Triangle strips generated by the algorithm

The algorithm produced 4 sequences representing triangle strips with an accumulated number of indices of 19. An indexed triangle mesh would have required 3(11) indices, to a total of 33. Thus, the objective function returns a compactness of 0.575758.

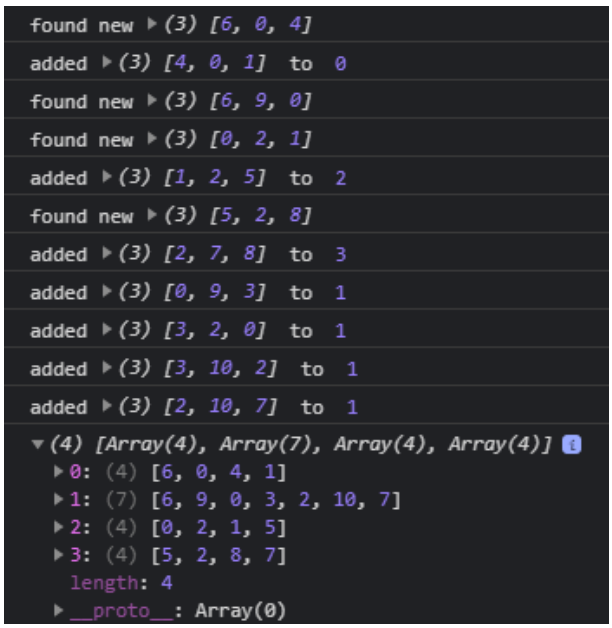

Figure 10: Decisions made by the algorithm during runtime

V. Conclusion

The usage of greedy algorithms is sufficient to generate compact triangle strip meshes from indexed triangle meshes. There are many faults present in the author's implementation of the algorithm, notably the lacking in testing using real models which can be an aspect to improve in the future.

Acknowledgment *(Heading 5)*

The author would like to thank God for His blessings and guidance, granting the author the strength to finish this paper. The author would also like to thank his sister, Jo, and mother, for giving emotional support, love, and affection throughout the making of this paper. The author also thanks his lecturer Prof. Ir. Dwi Hendratmo Widyantoro, M.Sc, Ph.D. as the lecturer of both IF2211 – Strategi Algoritma, the subject giving this paper assignment, and also IF3260 – Grafika Komputer, the subject that gave the author inspiration and the basics of computer graphics used throughout this paper. Finally, the author would like to thank all the author's colleagues for all the support they had given.

References

[1] Shirley, Peter and Steve Marschner, Fundamentals of Computer Graphics, 3rd ed, 2009,

[2] Kaick, Oliver Matias van, Murillo Vicente Goncalvas da Silva, and Helio Pedrini, "Efficient Generation of Triangle Strips from Triangulated Meshes", 2004, (https://www.researchgate.net/publication/221546772_Efficient_Generation_of_Triangle_Strips_from_Triangulated_Meshes), accessed on May 11th, 2021.

[3] Held, Martin, "Stripification of Polyhedral Models", 2021, (https://www.cosy.sbg.ac.at/~held/projects/strips/strips.html), accessed on May 11th, 2021.

[4] Ovsjanikov, Maks, “Triangle Mesh Processing”, (http://www.lix.polytechnique.fr/~maks/Verona_MPAM/TD/TD2/), accessed May 11th, 2021.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 April 2021

I Gede Govindabhakta