Geometry and Transformations

Slides by Sarah Jobalia and Kevin Li, with modifications by Kate Baker

Revisiting Questions

Revisiting Questions

- 2D vs. 3D animation – which is simpler?
  - 2D in general, although not as scalable
  - 3D more difficult, but typically better results for movies, games, VR applications, etc.

Revisiting Questions

- 2D vs. 3D animation – which is simpler?
  - 2D in general, although not as scalable
  - 3D more difficult, but typically better results for movies, games, VR applications, etc.
- Summer transcripts
  - Reach out to summer session / portal

Revisiting Questions

- 2D vs. 3D animation – which is simpler?
  - 2D in general, although not as scalable
  - 3D more difficult, but typically better results for movies, games, VR applications, etc.
- Summer transcripts
  - Reach out to summer session / portal
- Computer vision
  - Can talk about this more in advanced lecture
  - Check out OpenCV

Lecture Outline

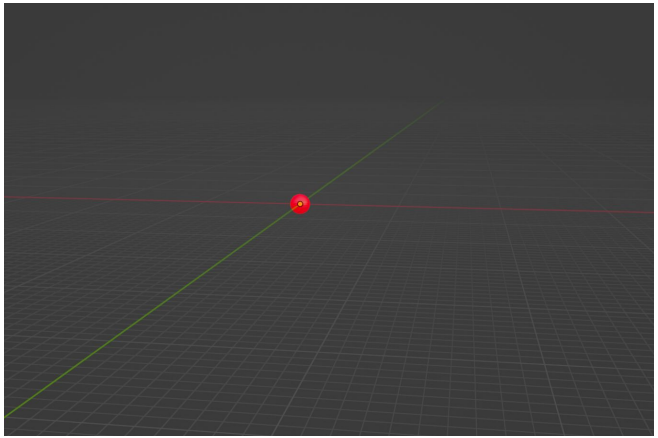
- Points and the GPU
- Triangles
- Subdivision
- Transformations

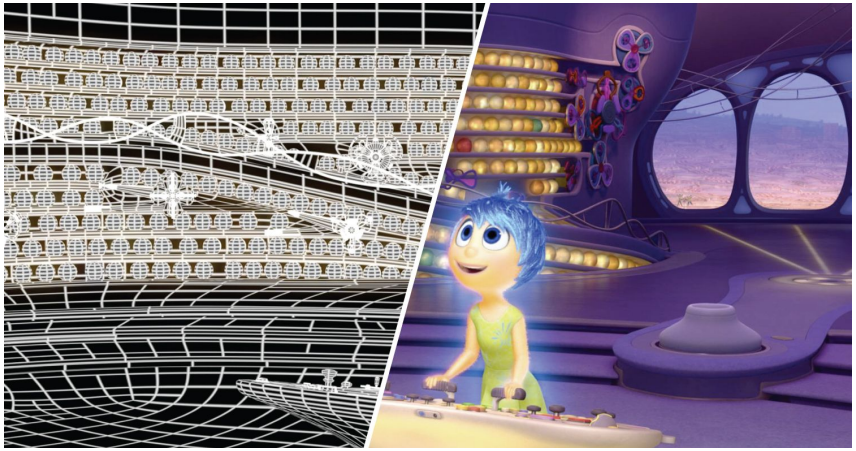
Lecture Outline

- Points and the GPU
- Triangles
- Subdivision
- Transformations

Points in Virtual Space

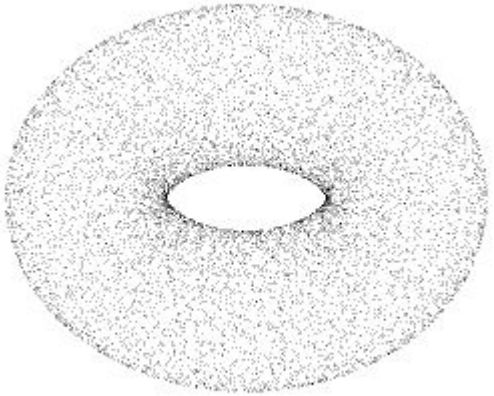
- 3D space: x, y, z coordinates
- Moving from points to coherent images





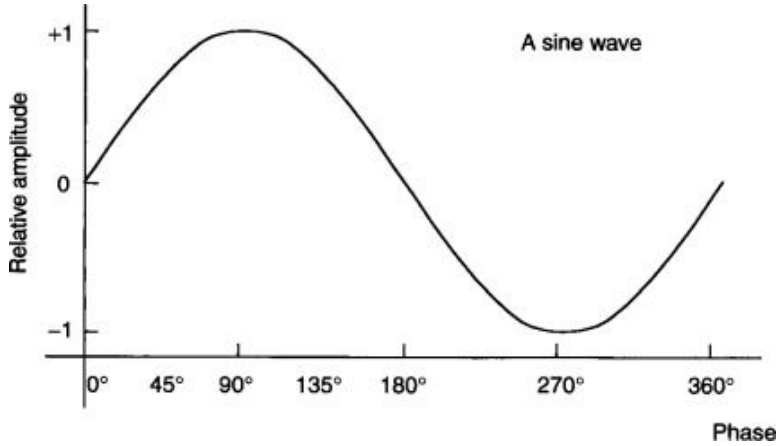
Point Clouds

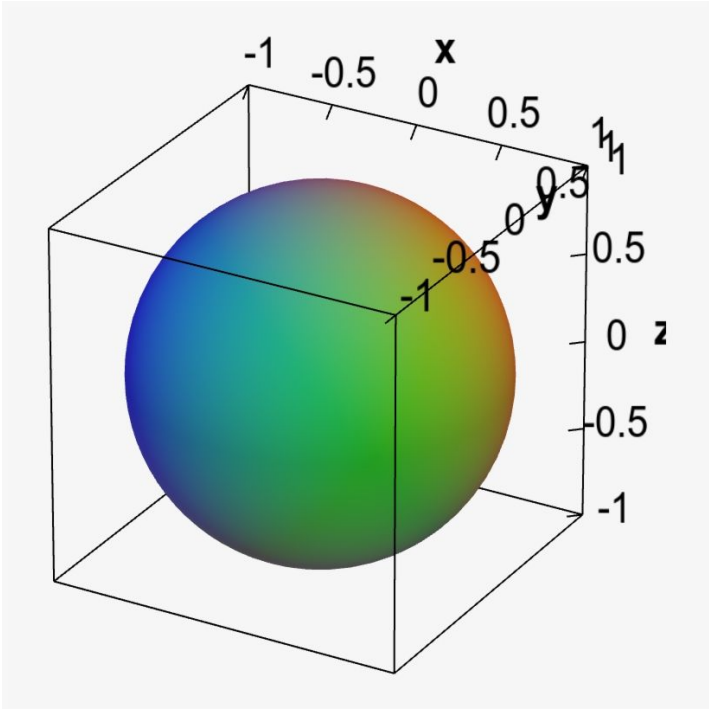
- Clusters of points



Implicit Surfaces

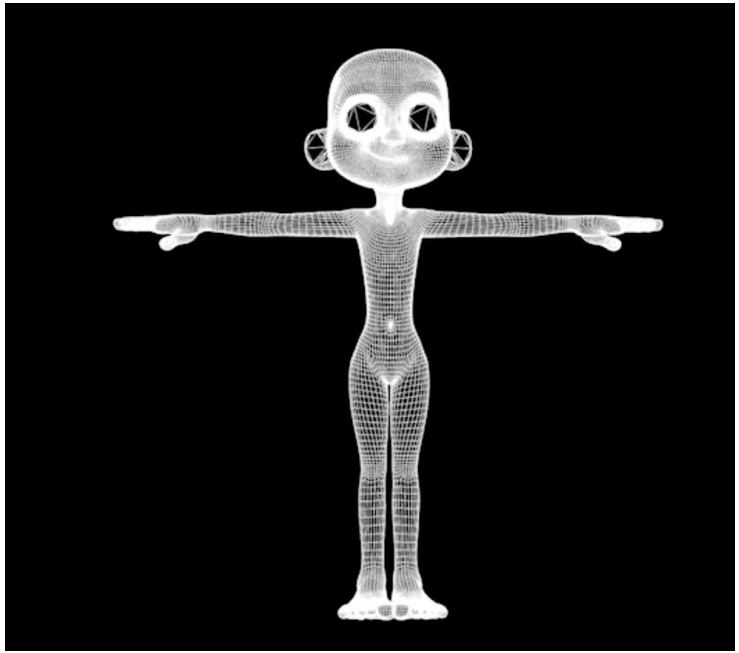
- Moving away from specific points

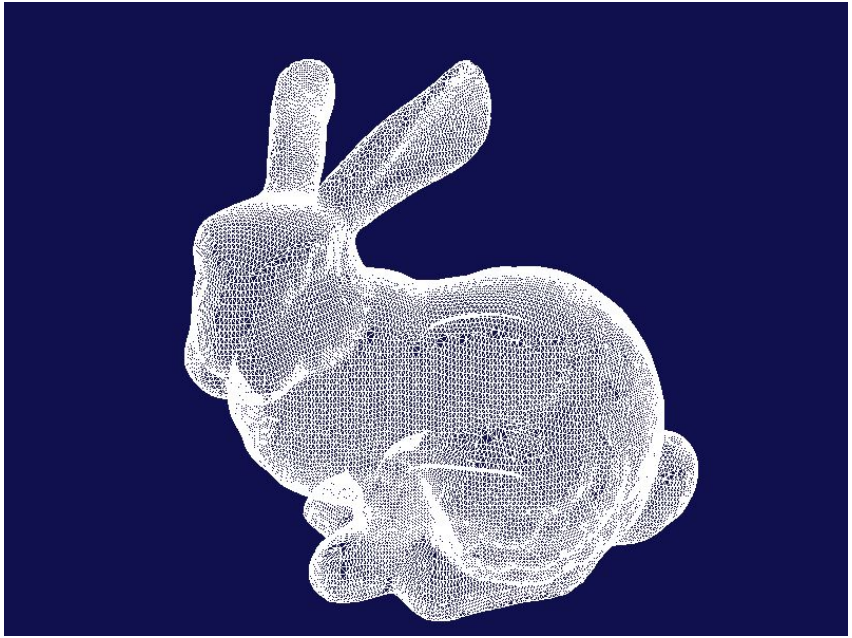




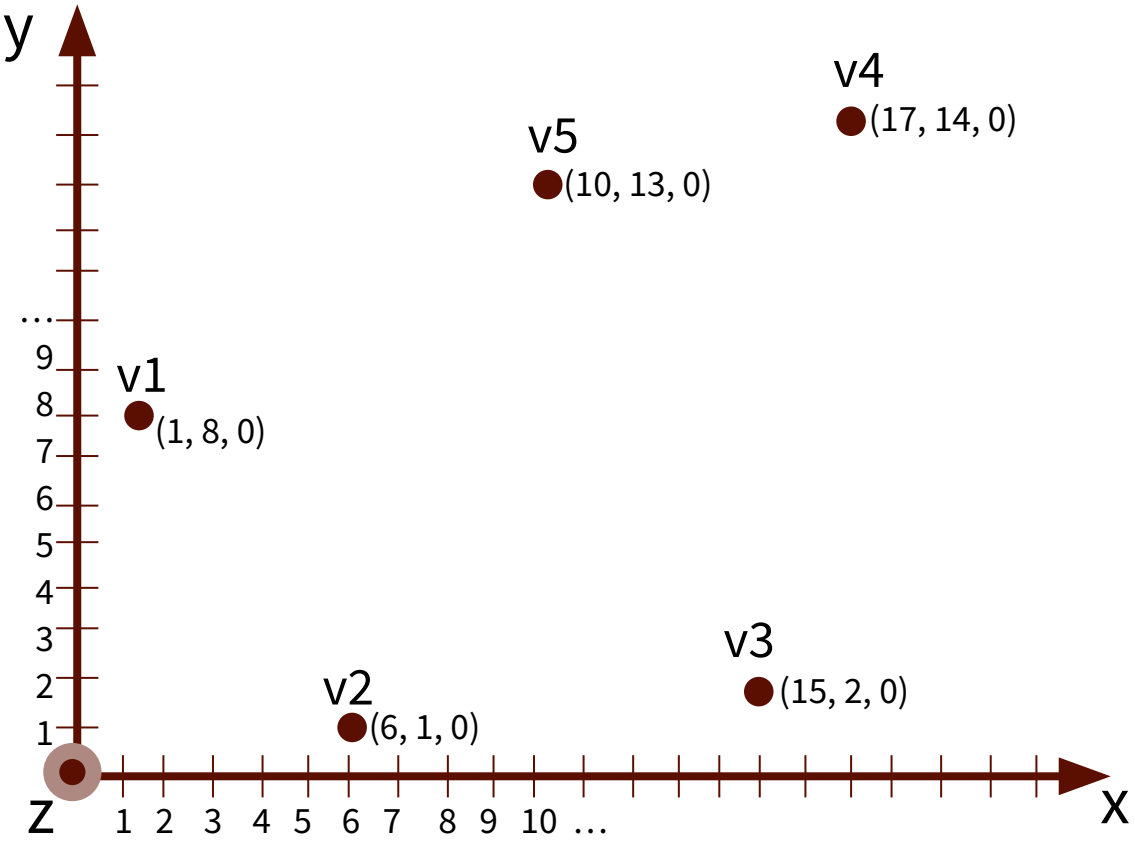
Mesh Surfaces

- Tool for representing complex objects

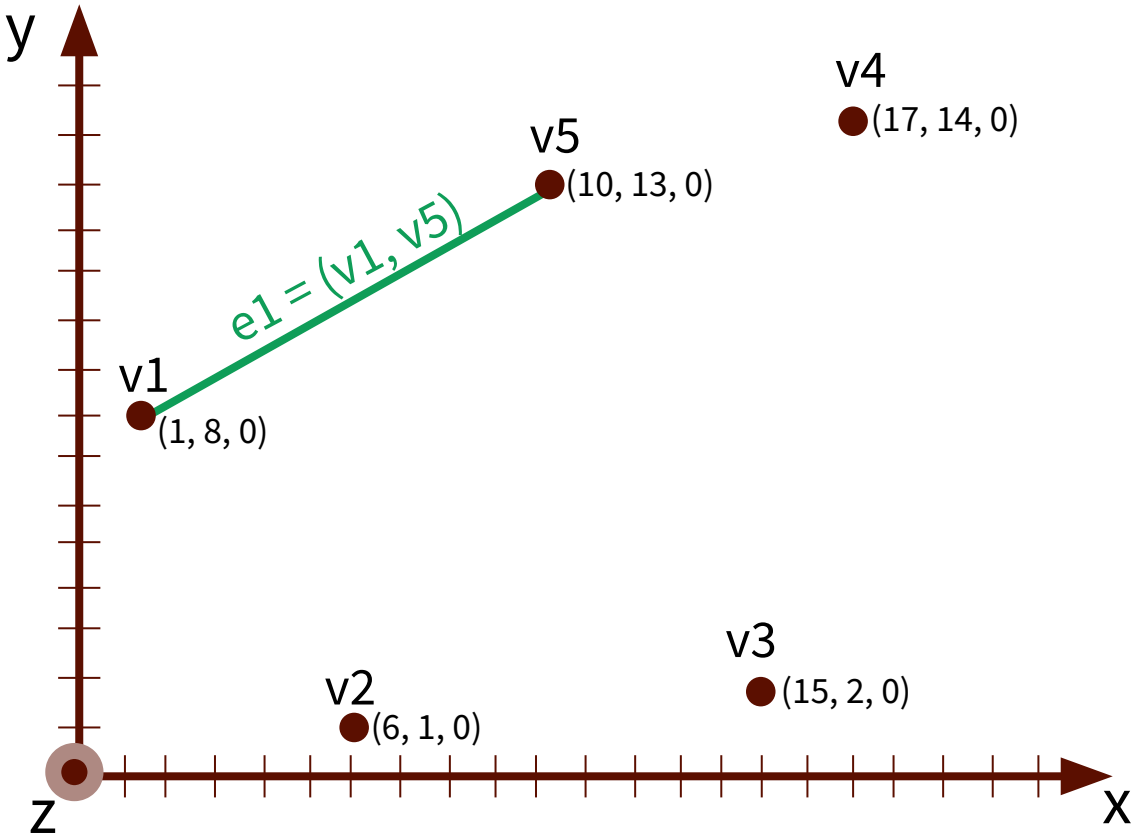




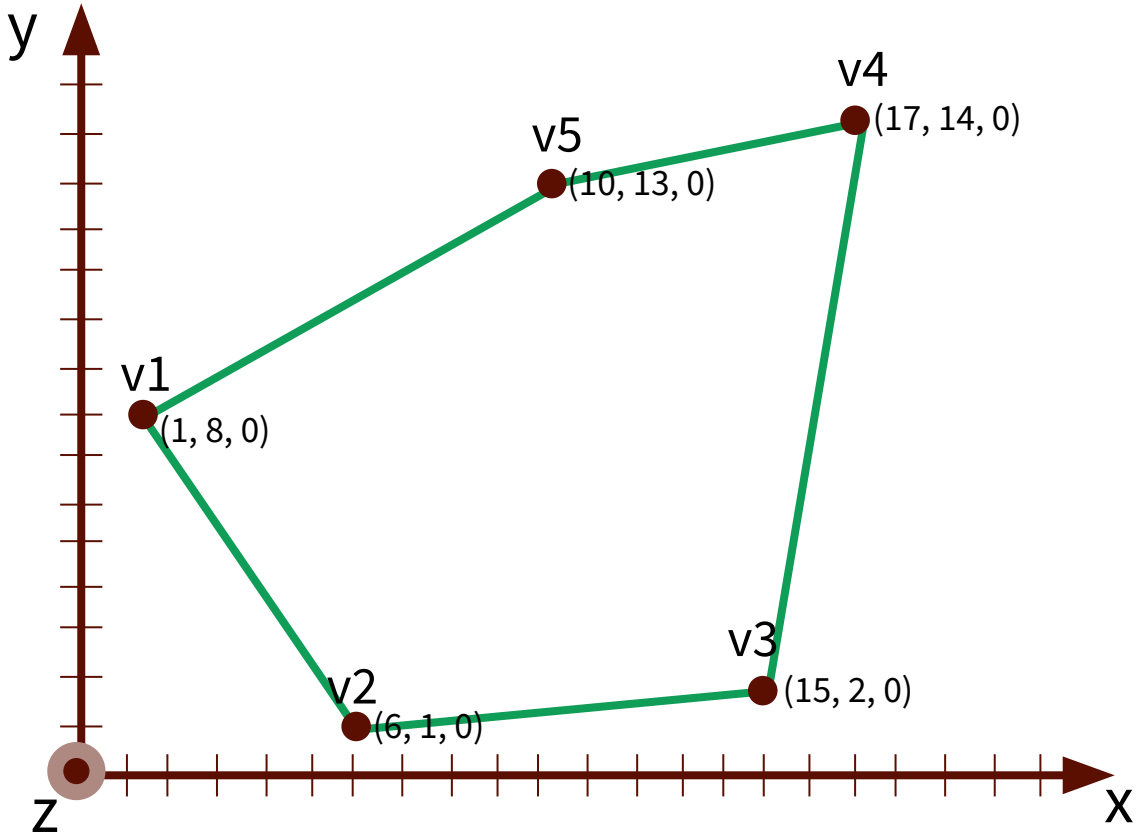
Points, Edges, and Faces



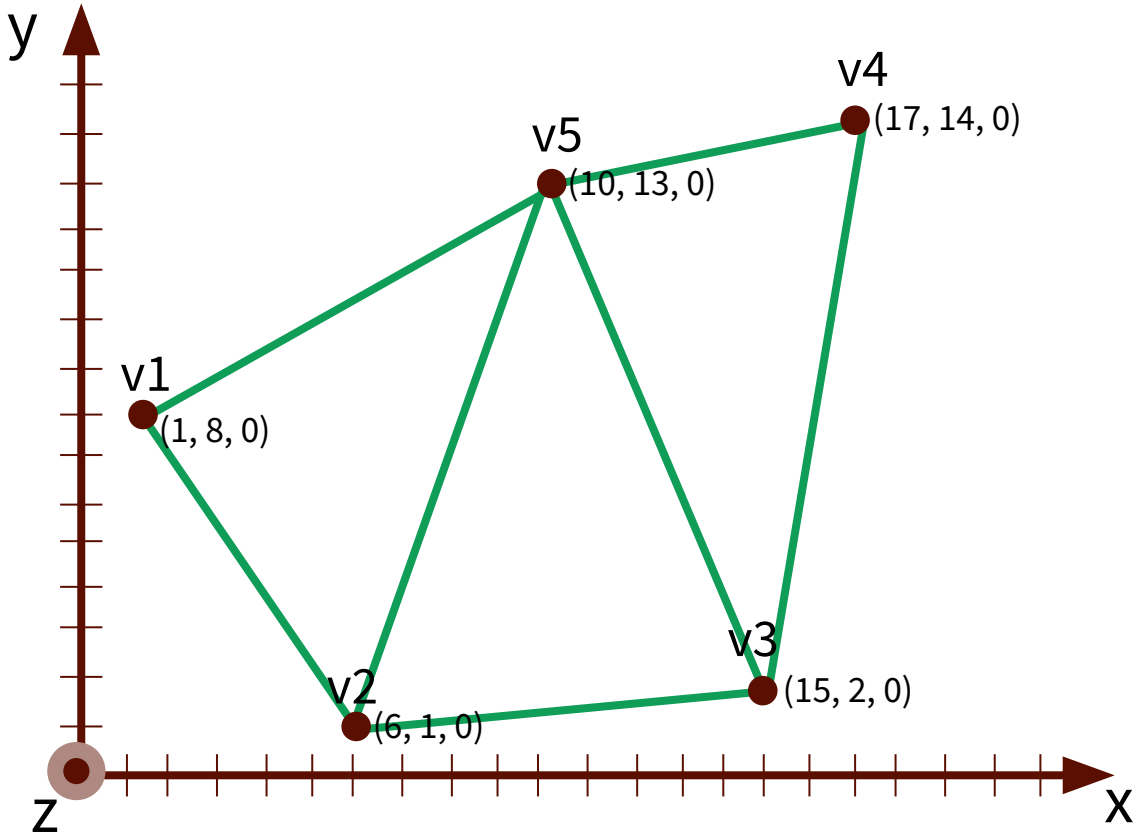
Points, Edges, and Faces



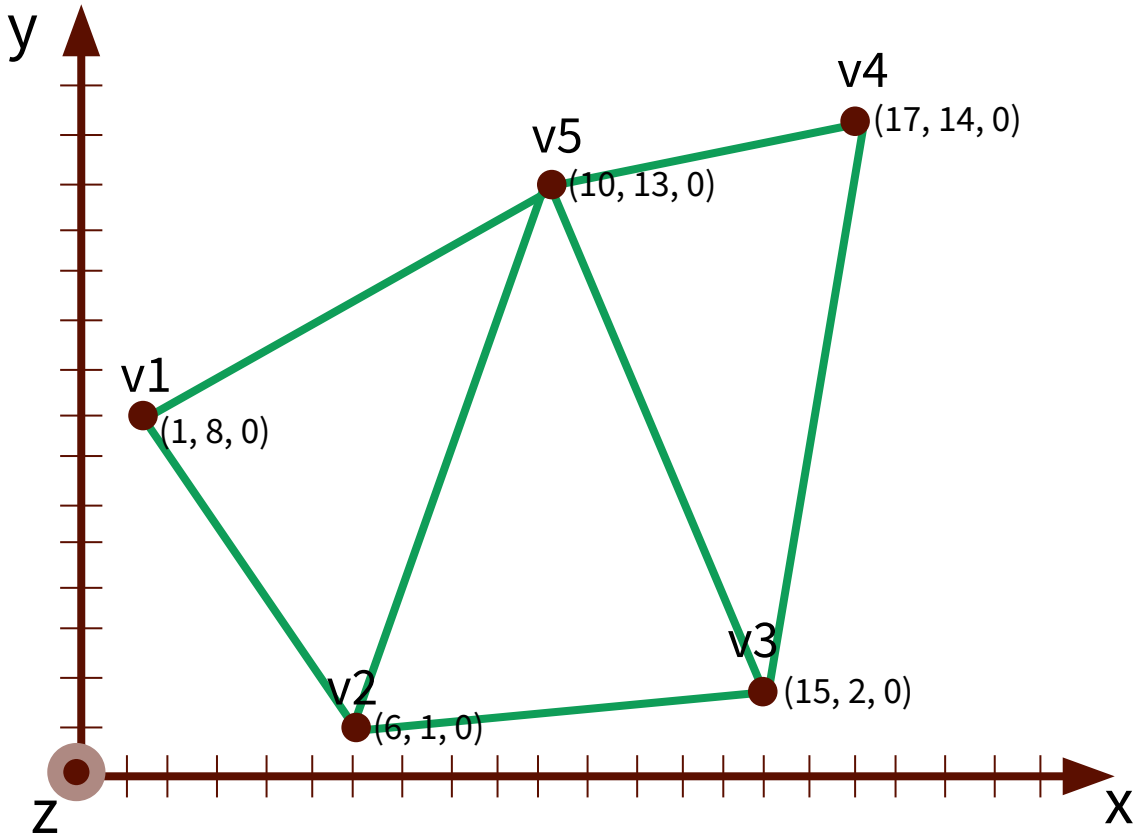
Points, Edges, and Faces



Points, Edges, and Faces



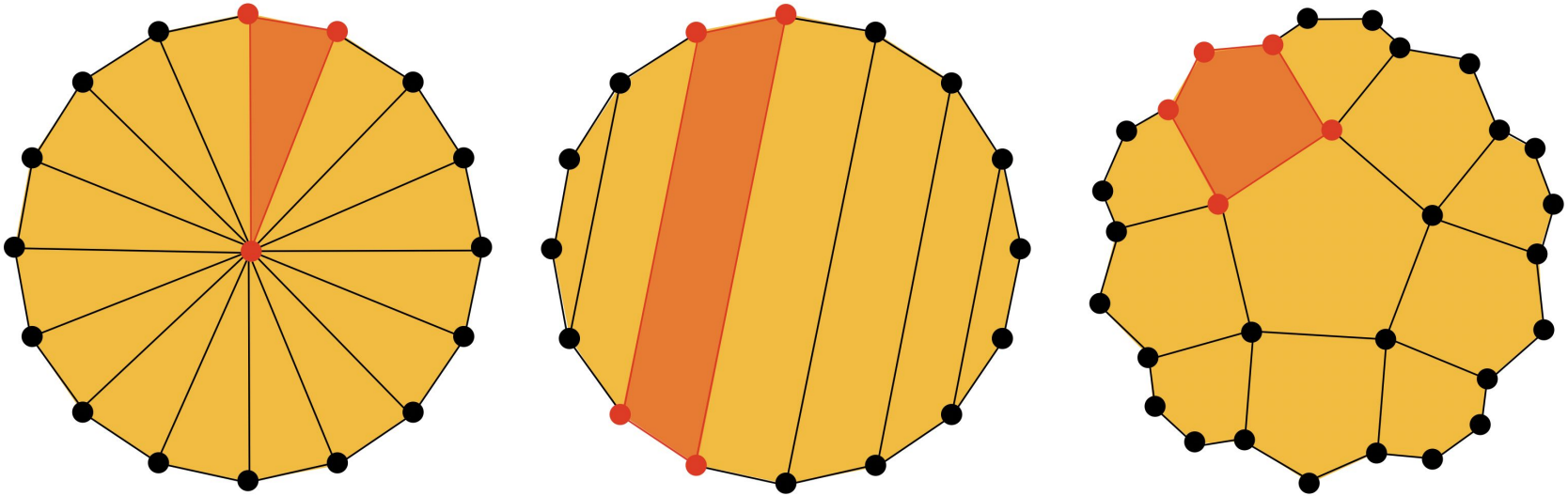
Points, Edges, and Faces



- Ideally, all of the faces should have the same number of vertices

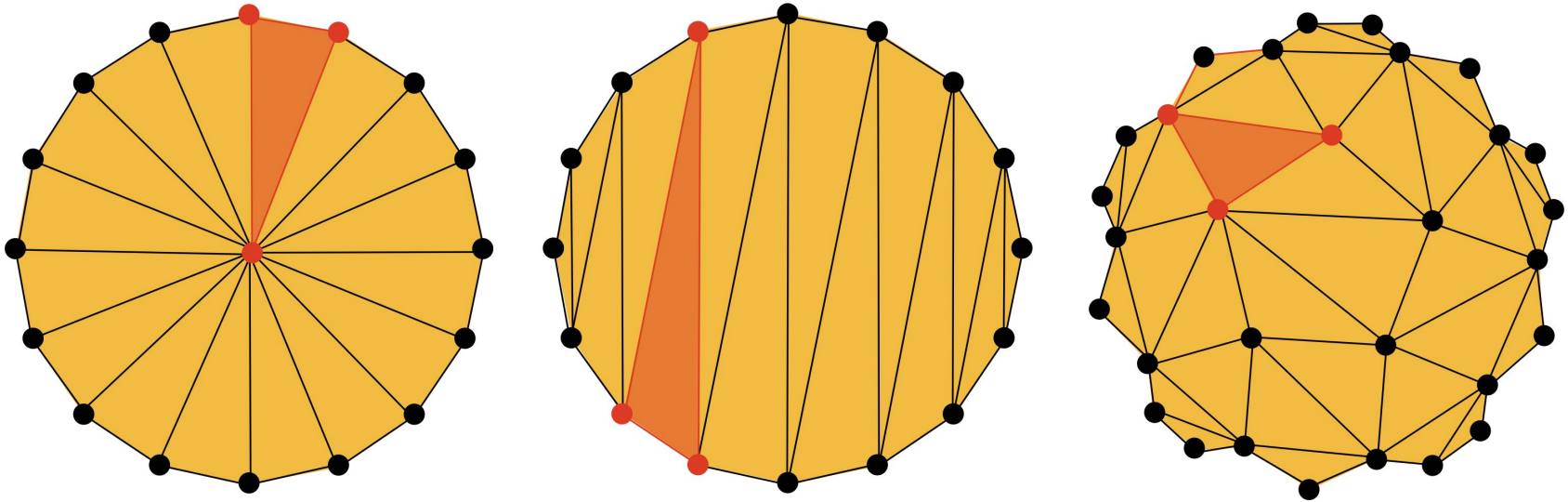
Points, Edges, and Faces

- These shapes have the same number of vertices
- But for the GPU, we should choose a single standard



Points, Edges, and Faces

- We choose the mighty triangle



Lecture Outline

- Points and the GPU
- Triangles
- Subdivision
- Transformations

Why Triangles?

- Easy to break other polygons into triangles
  - Complex objects are well approximated with triangles

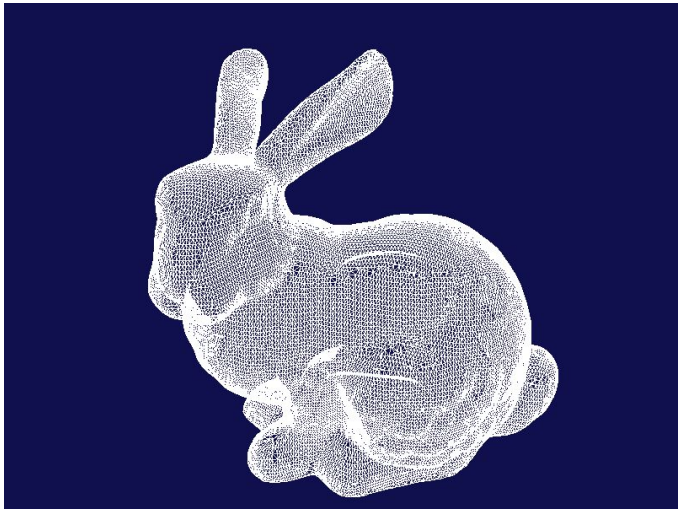
Why Triangles?

- Easy to break other polygons into triangles
  - Complex objects are well approximated with triangles
- Guaranteed planar

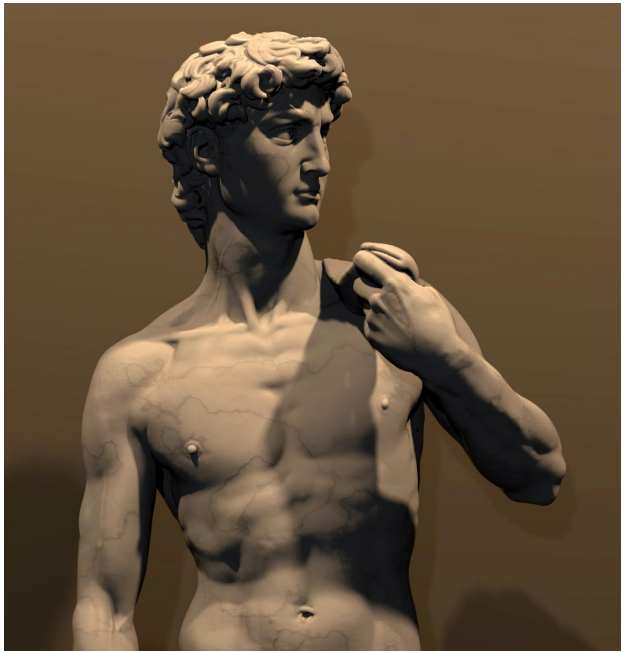
Why Triangles?

- Easy to break other polygons into triangles
  - Complex objects are well approximated with triangles
- Guaranteed planar
- Geometric transformations only need to be applied to vertices

Why Triangles?



Stanford Bunny
69,451 triangles



David (Digital Michelangelo Project)
56,230,343 triangles

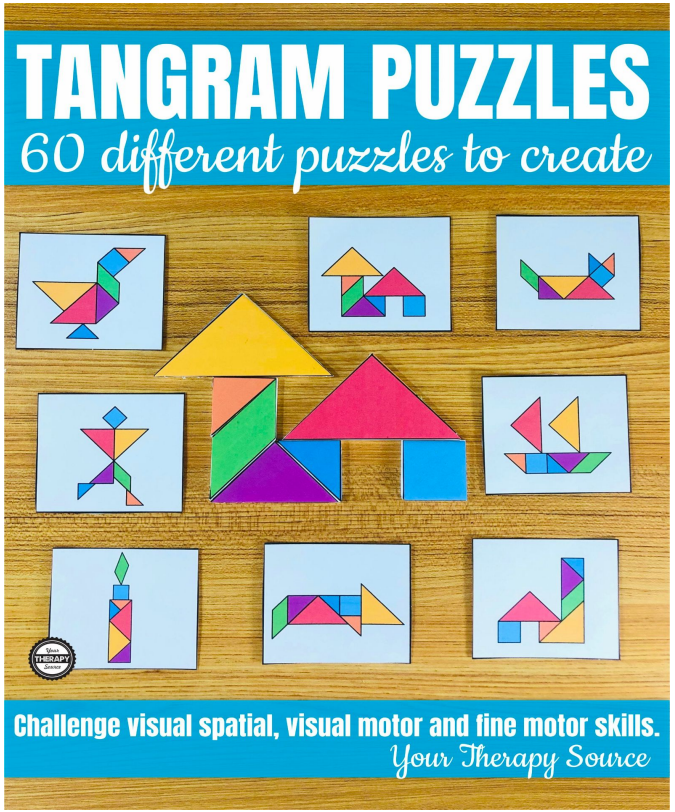
Why Triangles?

- Easy to break other polygons into triangles
- Guaranteed planar
- Complex objects are well approximated with triangles
- Geometric transformations only need to be applied to vertices

- Not everything though…
  - E.g. fluid simulation

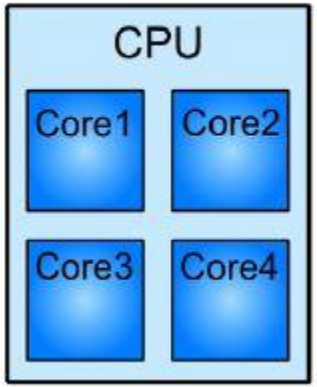
Why Triangles?

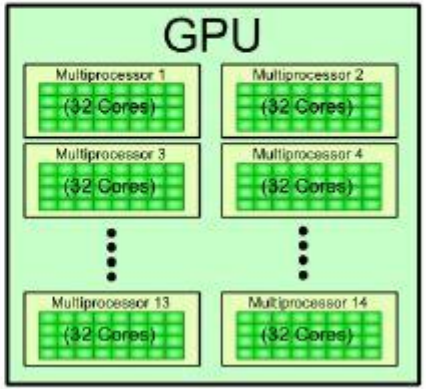
- Can optimize and specialize the geometry pipeline for 1 shape
- Software and algorithms can be optimized
- Hardware (e.g. GPUs) can be specialized

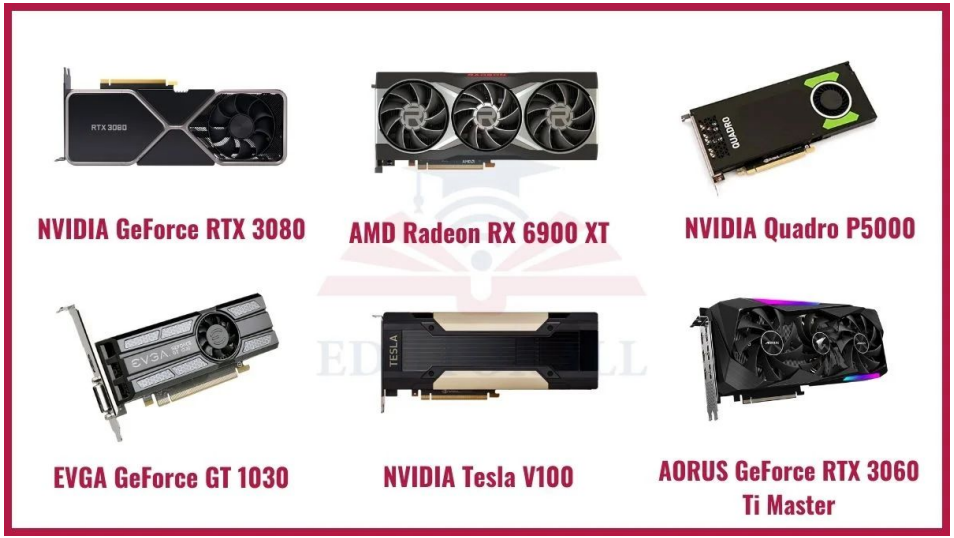


The GPU

- GPU = perfect for parallel operations
- Involves the same type of instruction executed many times at once
  - E.g. triangles!
- CUDA language

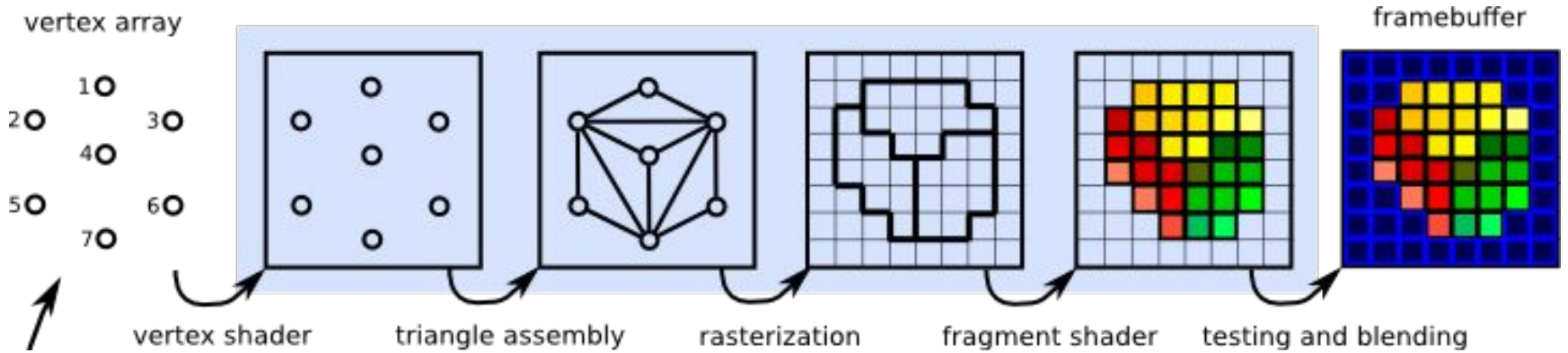






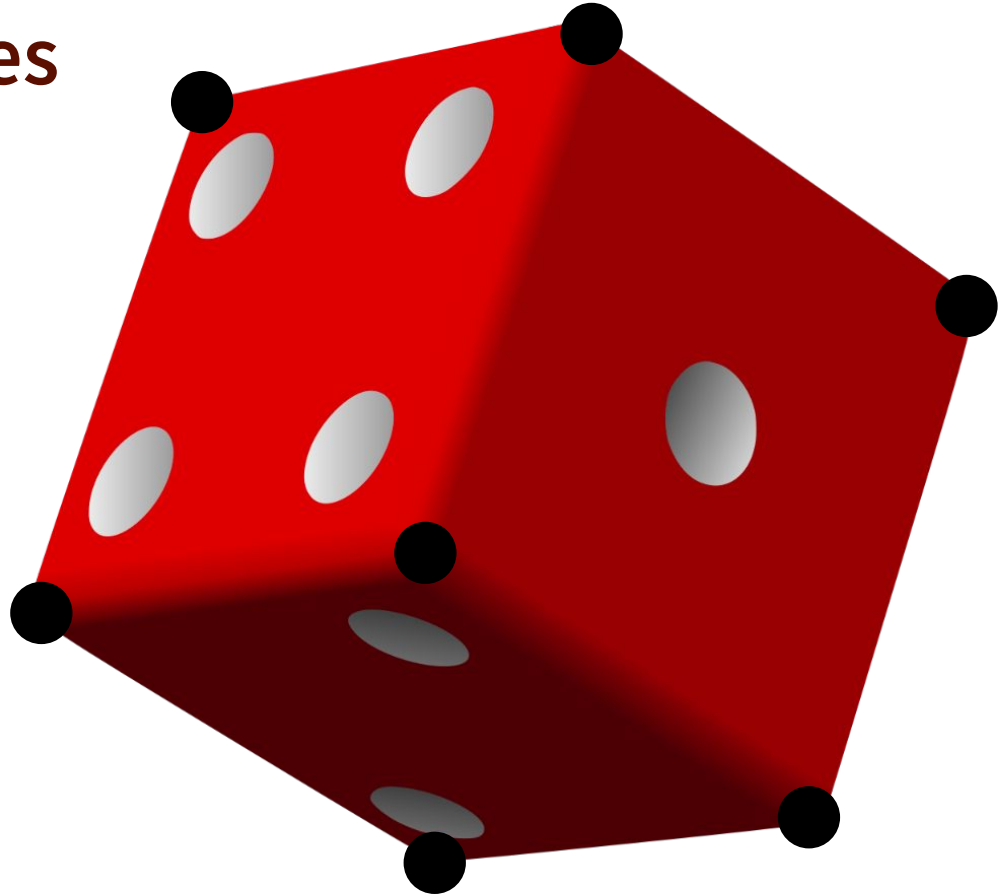
OpenGL and the Graphics Pipeline

- OpenGL
  - Real-time scanline renderer
  - Drawing API for 2D/3D graphics
  - Optimized for triangles

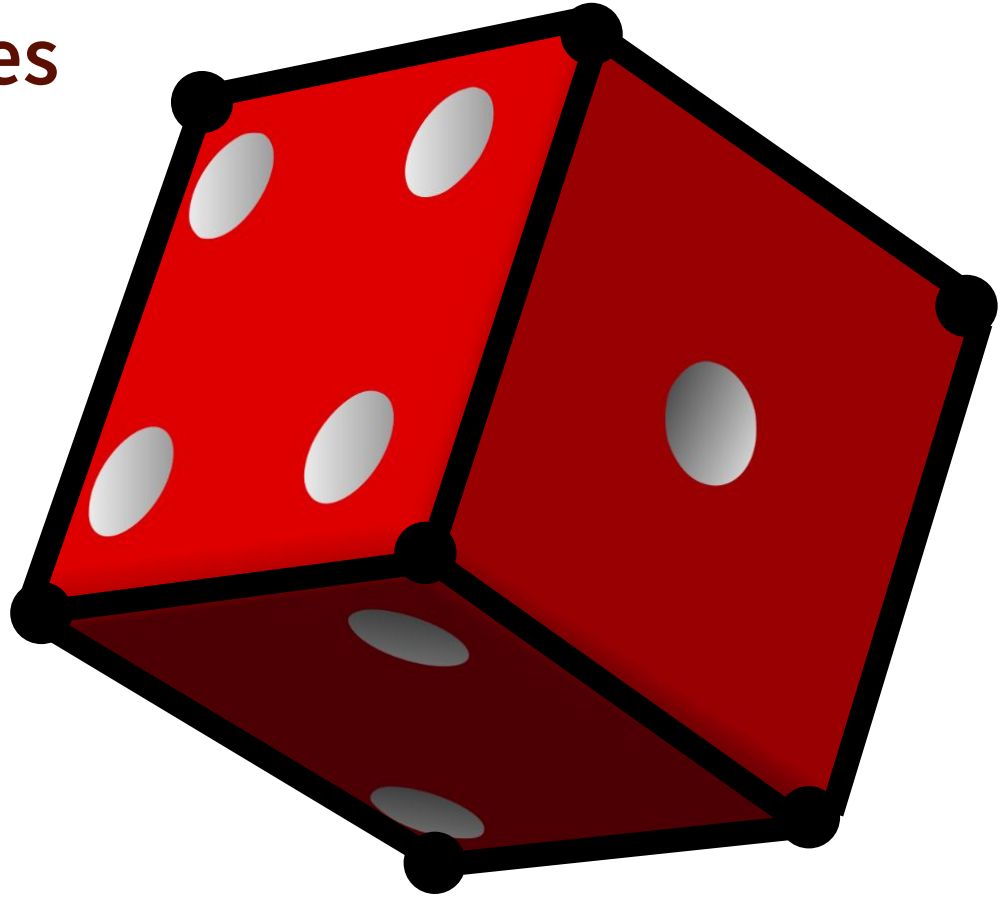


Ex: Triangles

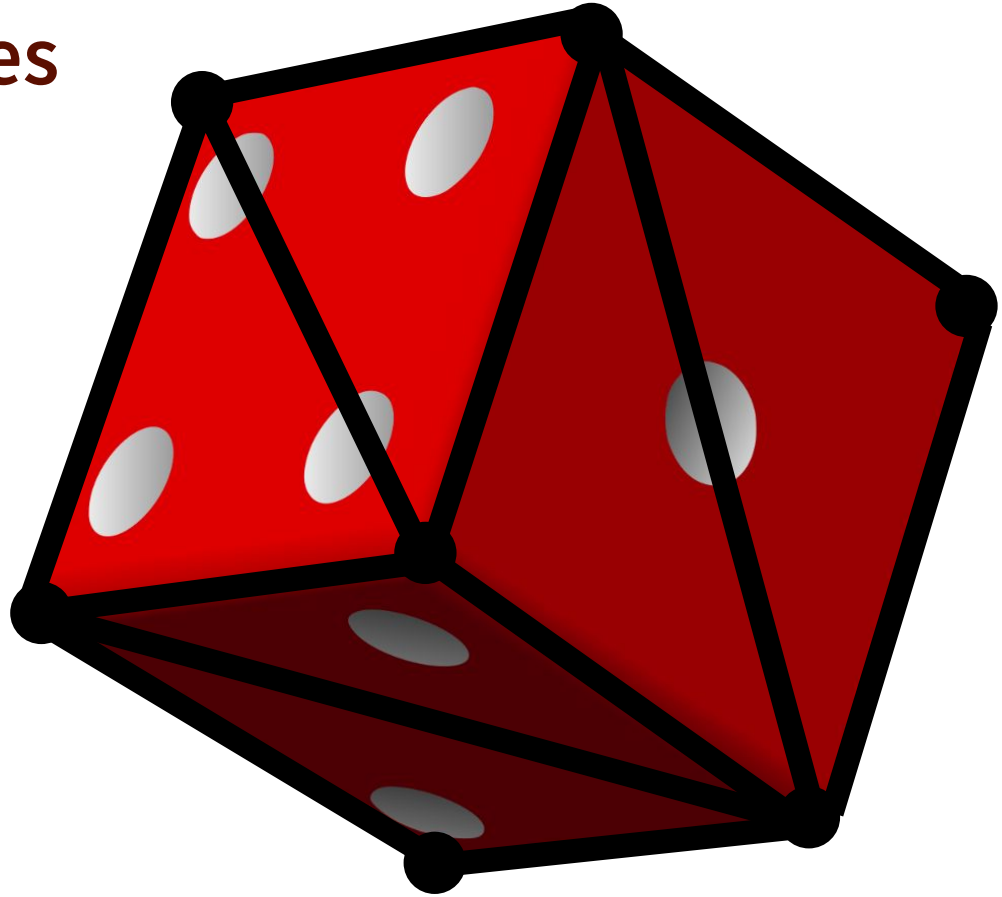
Ex: Triangles



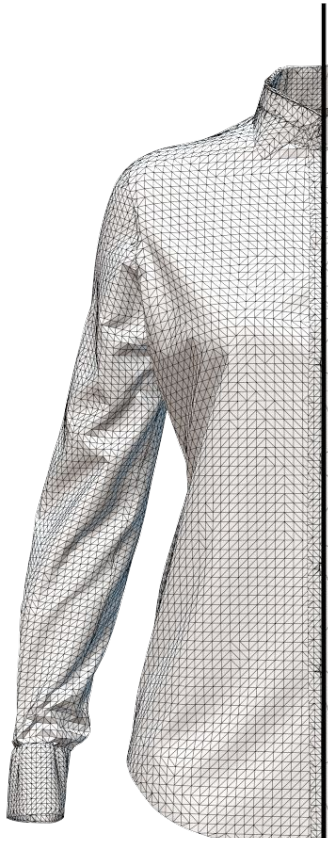
Ex: Triangles



Ex: Triangles

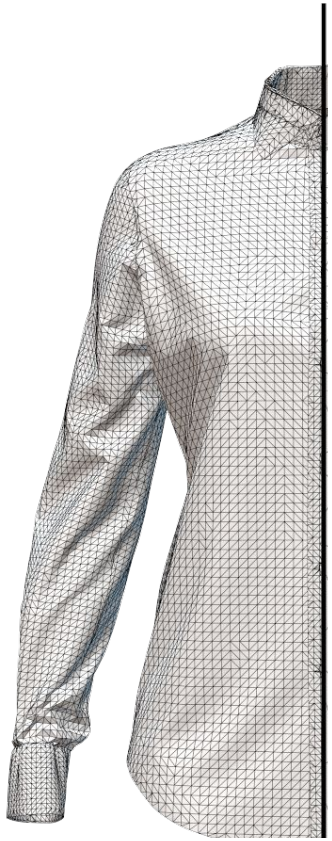


Ex: Triangles



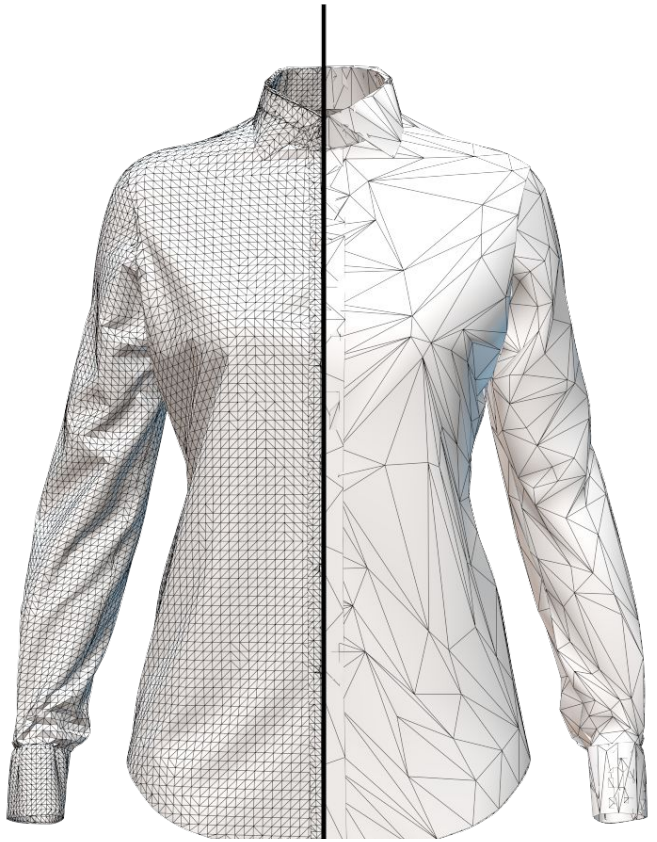
Ex: Triangles

- Tradeoff between density of points and computation



Ex: Triangles

- Tradeoff between density of points and computation



OBJ Files

- (One of) universal geometry definition file formats used by graphics applications

OBJ Files

- (One of) universal geometry definition file formats used by graphics applications

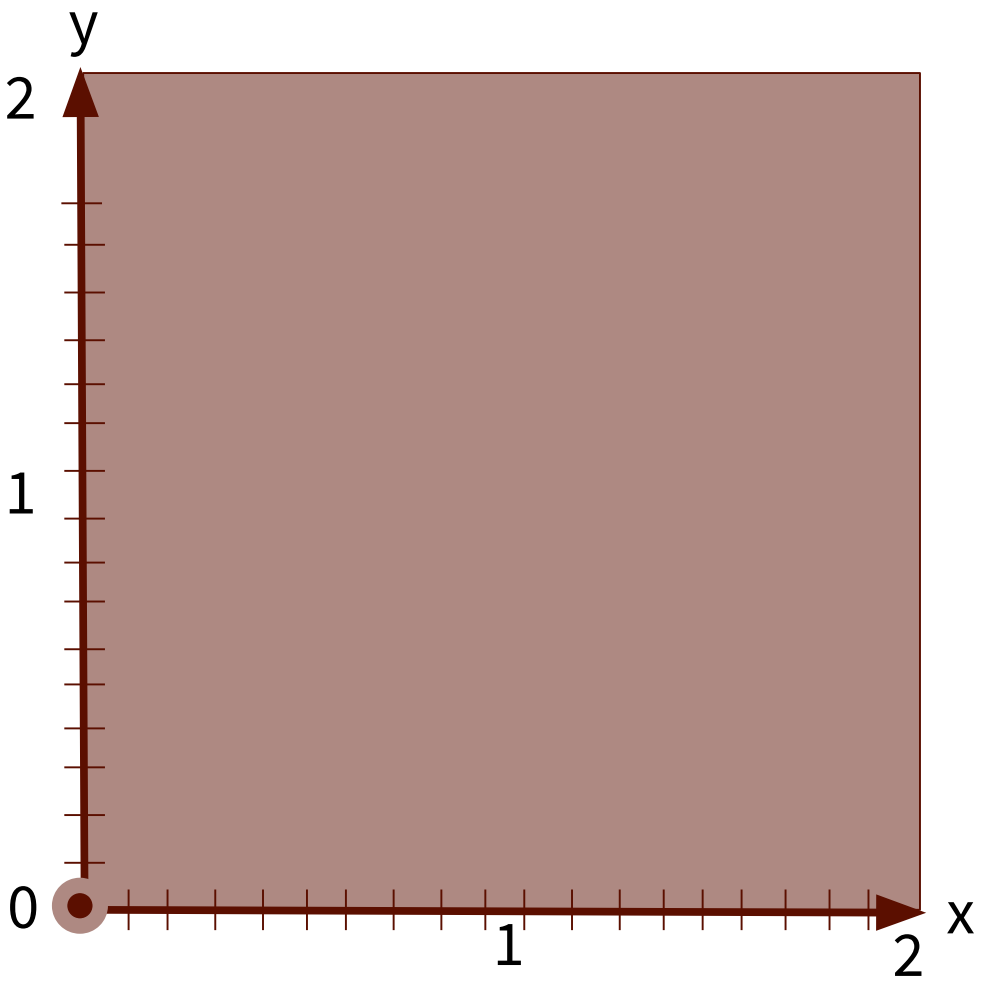
- Most basic form: list of vertices and faces
- “v” lines denote x, y, z coordinates of vertices
- “f” lines denote indices of vertices for the face

- Indices 1 indexed

- Also supports other polygon meshes

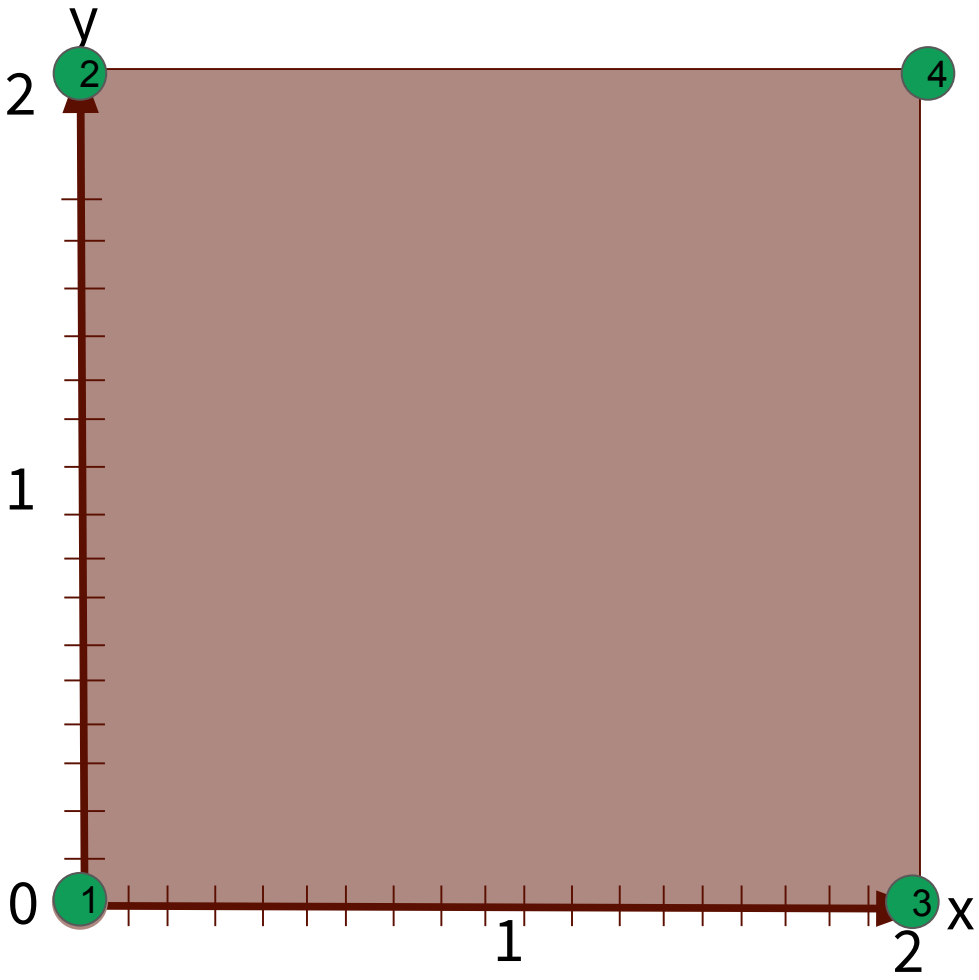
```
v x1 y1 z1
v x2 y2 z2
v x3 y3 z3
...
v xm ym zm
f face0v1 face0v2 face0v3
f face1v1 face1v2 face1v3
f face2v1 face2v2 face2v3
...
f facenv1 facenv2 facenv3
```

Ex: OBJ Files



Ex: OBJ Files

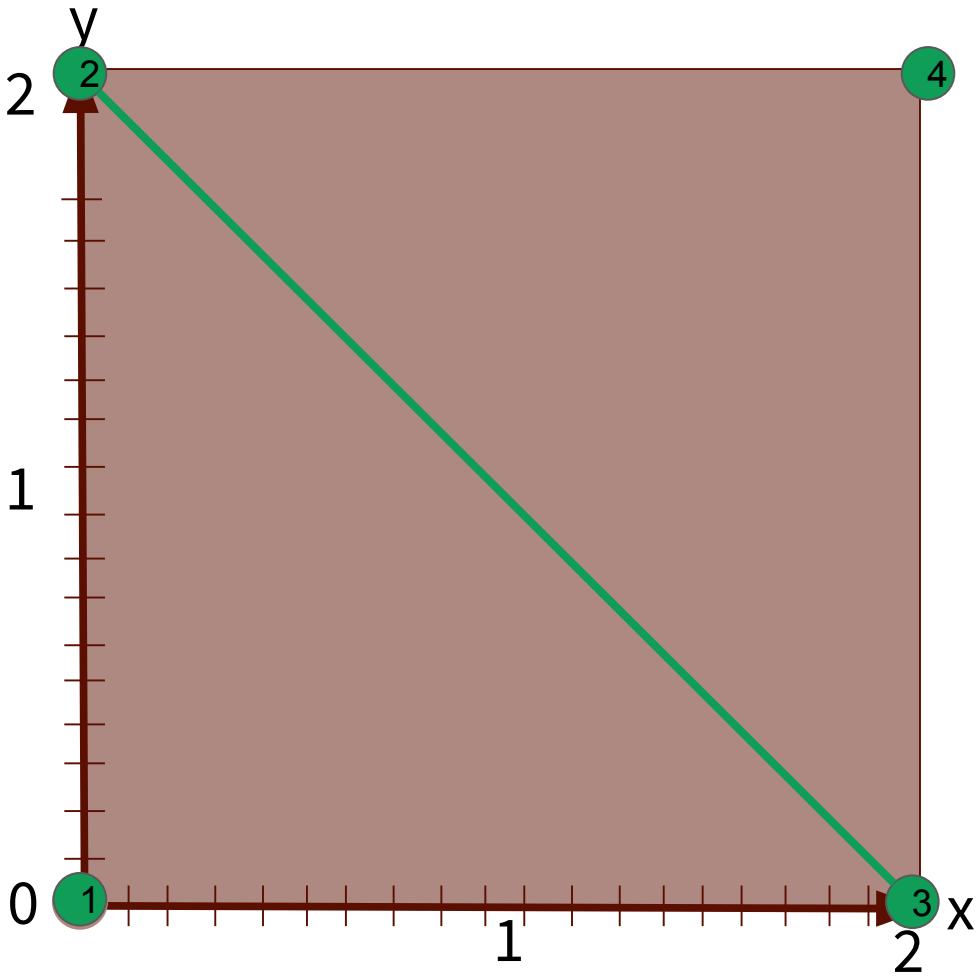
```
v 0 0 0
v 0 2 0
v 2 0 0
v 2 2 0
```



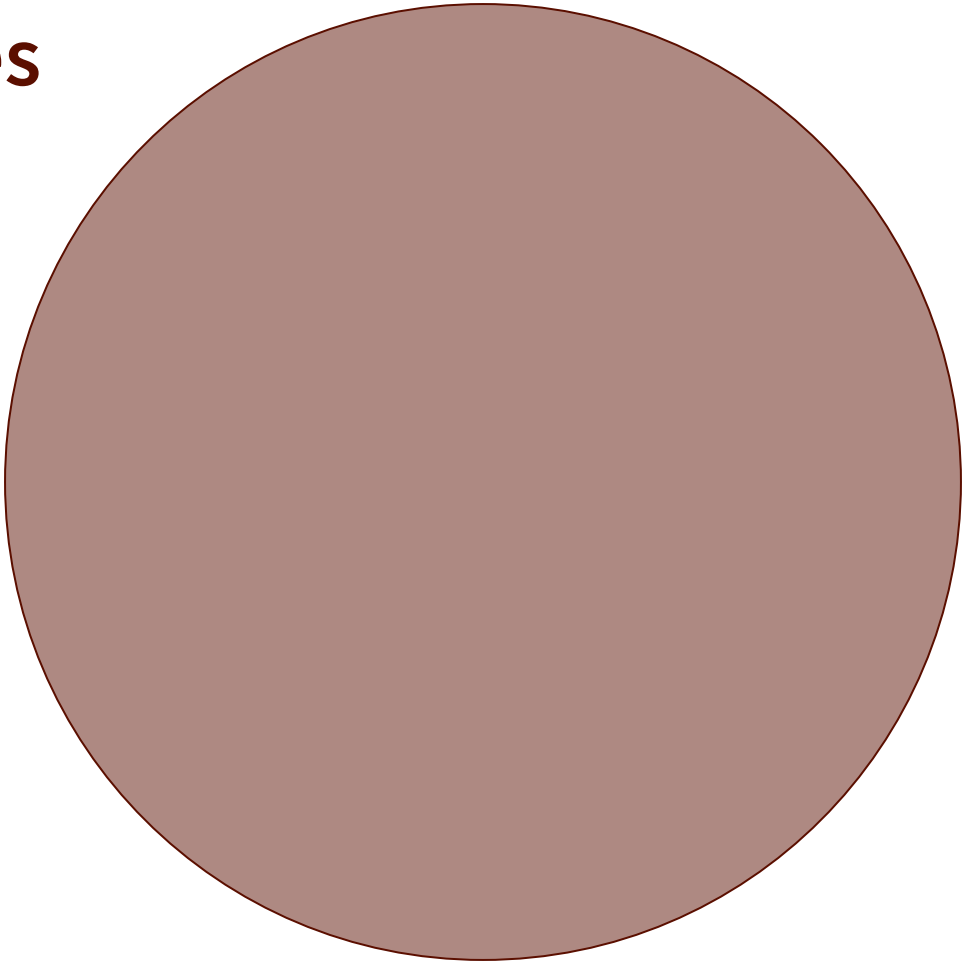
Ex: OBJ Files

```
v 0 0 0
v 0 2 0
v 2 0 0
v 2 2 0

f 1 2 3
f 2 3 4
```

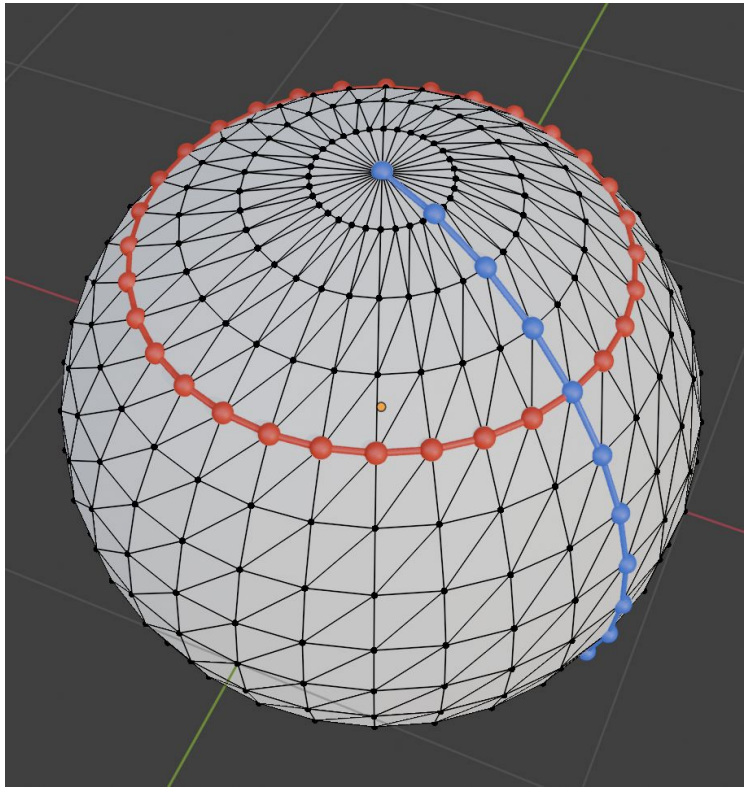


Ex: OBJ Files

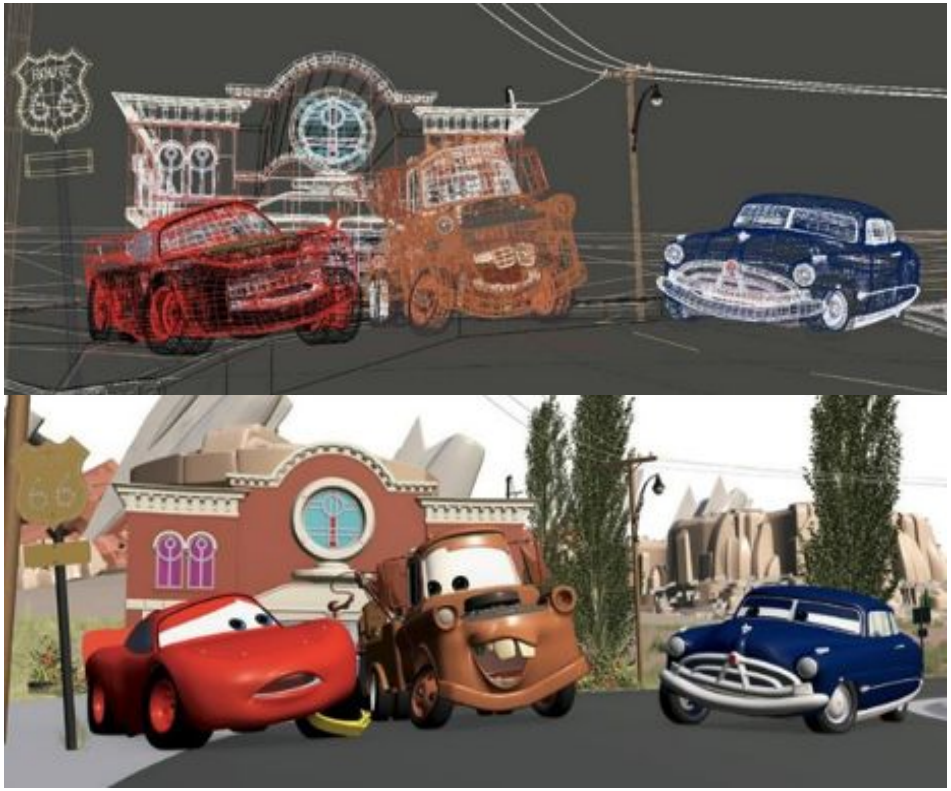


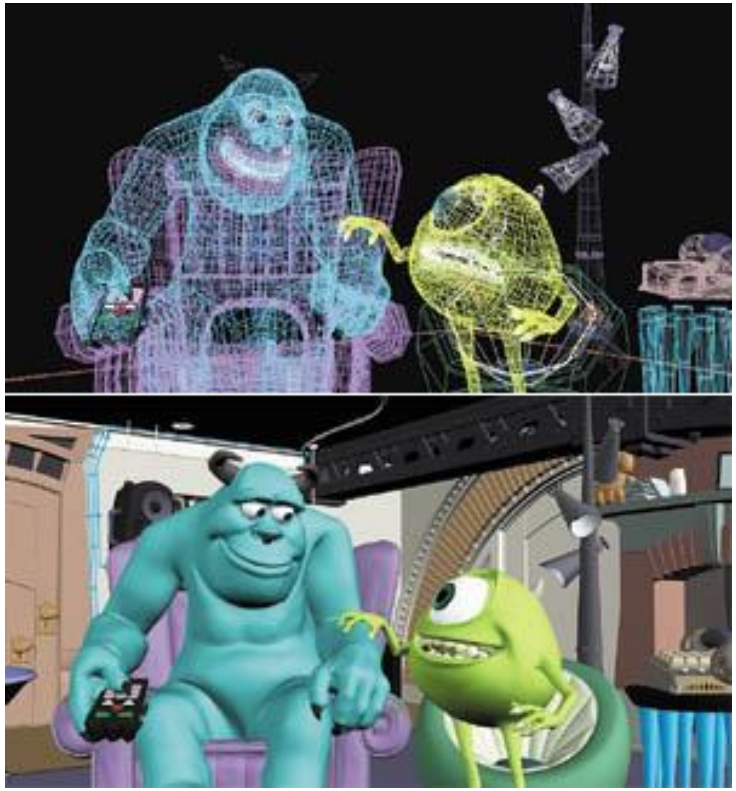
Ex: Tessellating a Sphere

- (Exercise on homework)

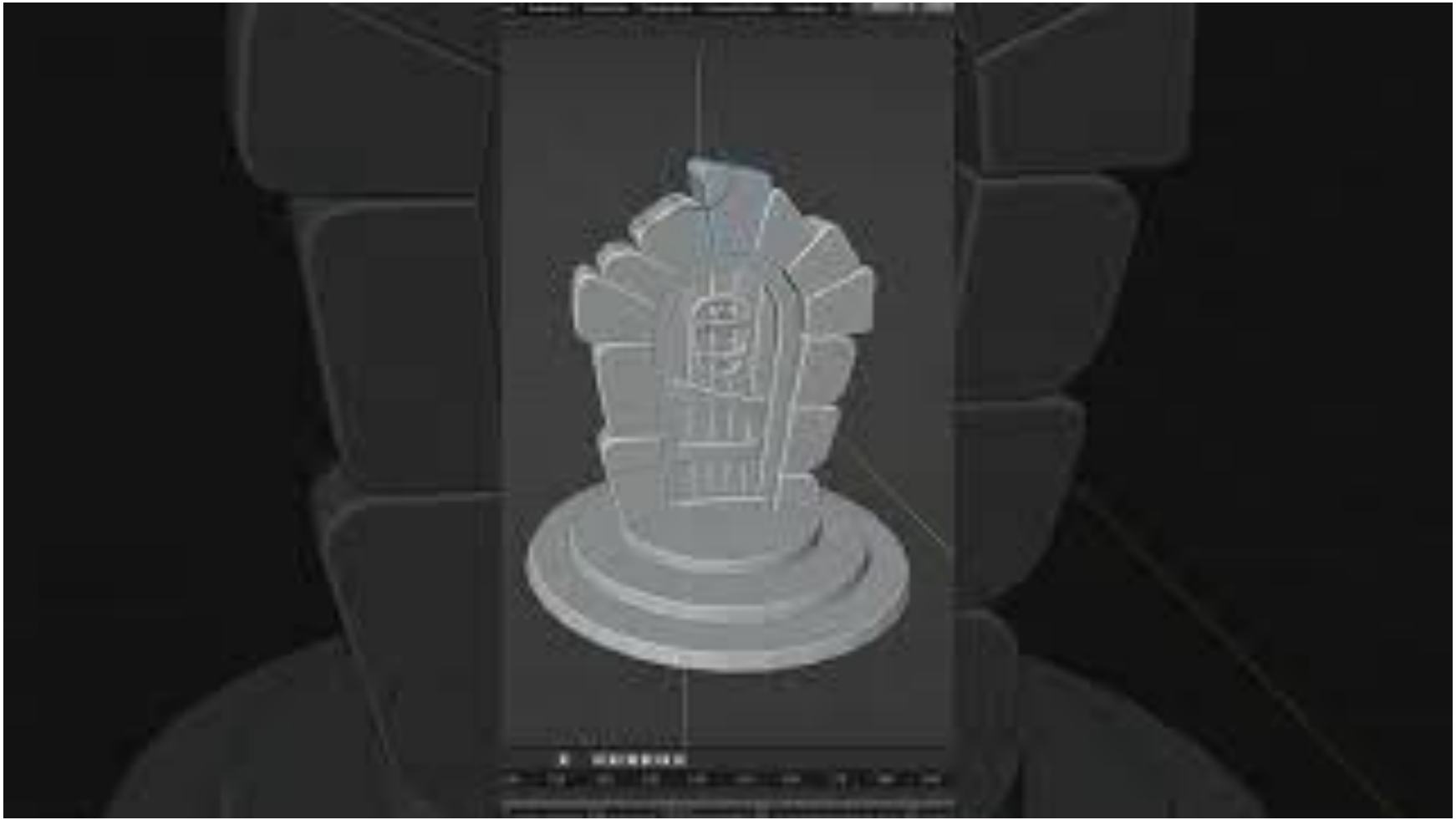


Ex: Movies

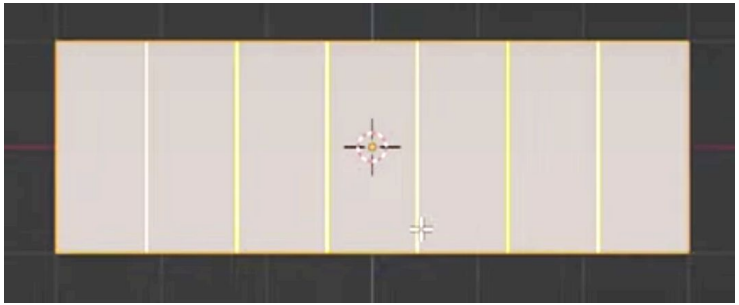


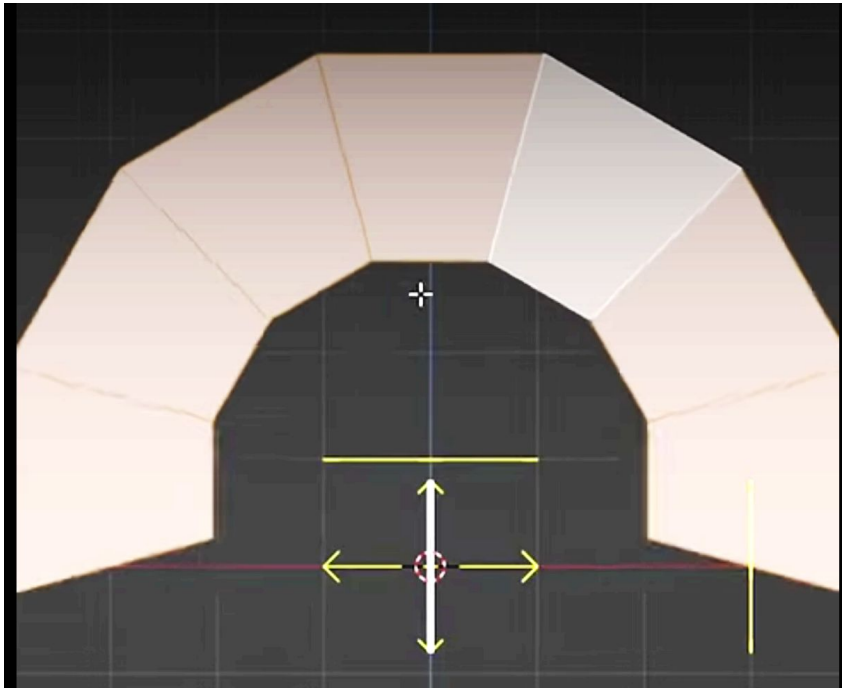


Ex: More Objects



Ex: More Objects





Activity: Make your own Object

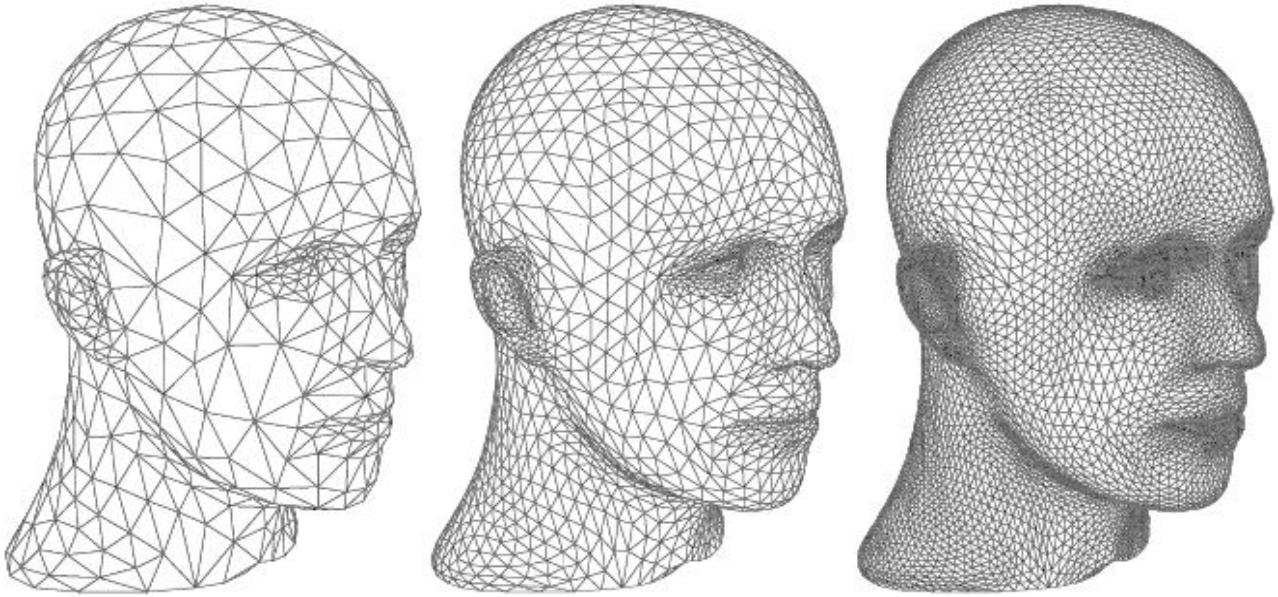
- (Ideally) get into groups
- Find an object you might want to reproduce virtually
- Discuss where vertices and edges might go
  - Think about what shapes the object is made out of

Lecture Outline

- Points and the GPU
- Triangles
- Subdivision
- Transformations

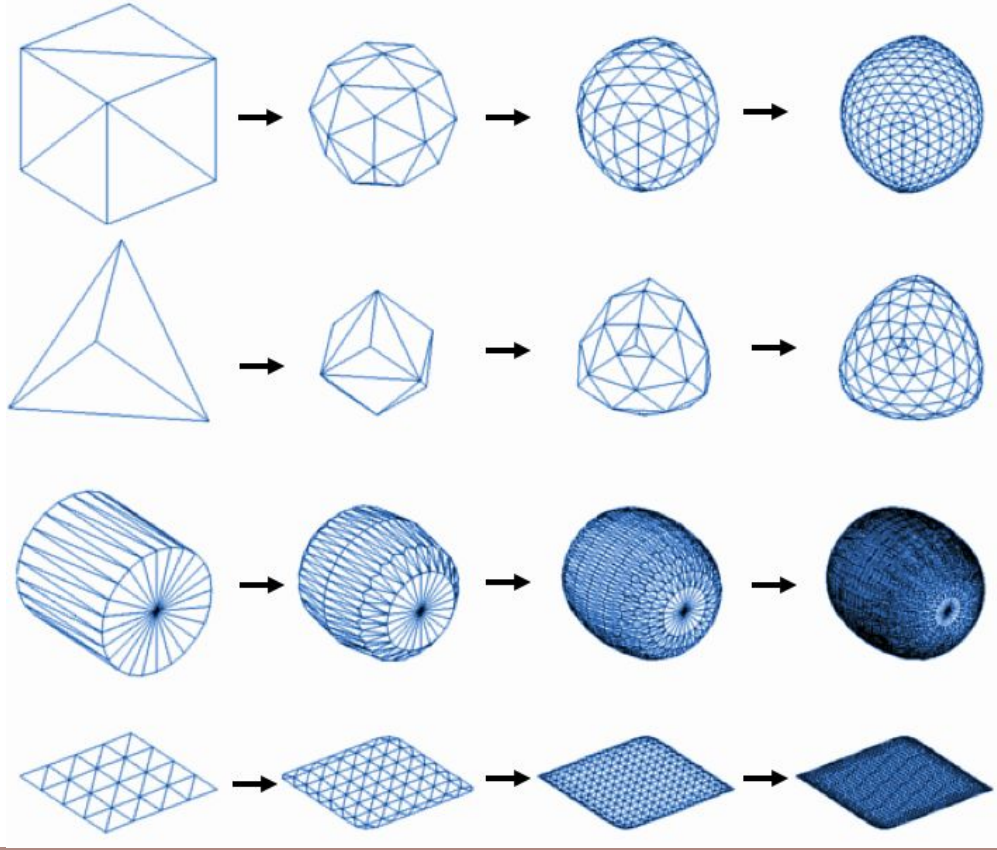
Subdivision

- A procedural algorithm
- Automatically generate finer/smoother mesh from a coarser mesh



Subdivision of Surfaces

- Using subdivisions to make shapes out of initial design



Charles Loop

Smooth Subdivision Surfaces Based on Triangles

by

Charles Teorell Loop

A thesis submitted to the faculty of
the University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mathematics

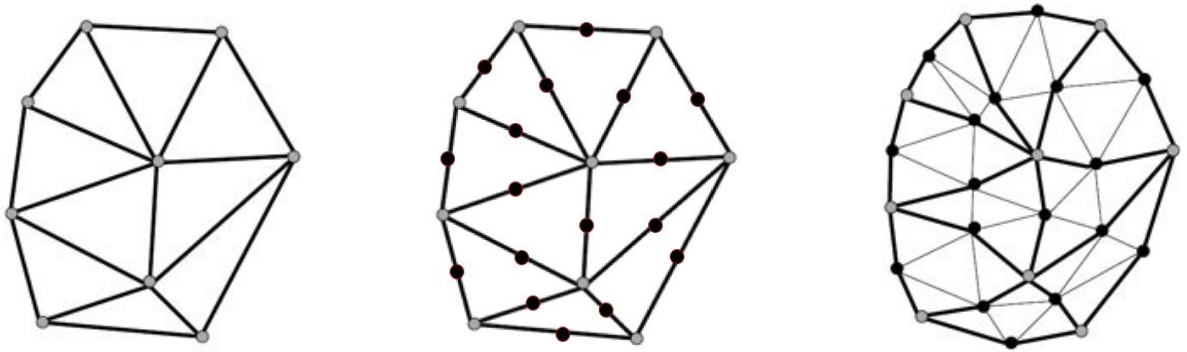
The University of Utah

August 1987

2478 citations (and counting) MS thesis!

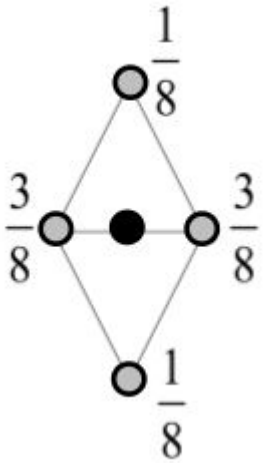
Loop Subdivision

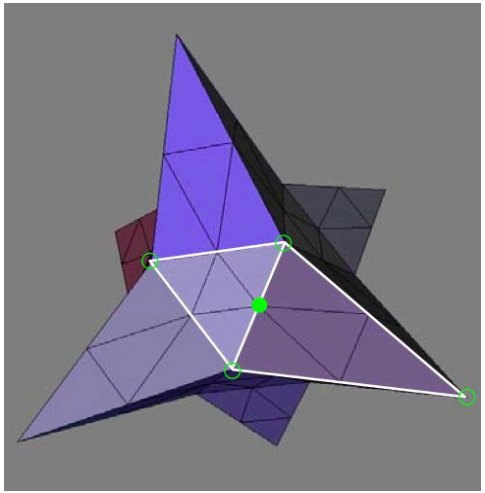
- Process
  - Subdivide each triangle into 4 sub-triangles
  - Move both the old/new vertices
  - Repeat (if desired)
- C^2 continuity almost everywhere
  - Except at some extraordinary vertices where it's only C^1



Loop Subdivision: Moving New Vertices

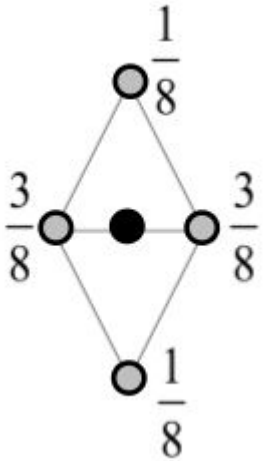
- Perturb the position of each new vertex (black) using a weighted average of the four nearby original vertices (grey)





Loop Subdivision: Moving New Vertices

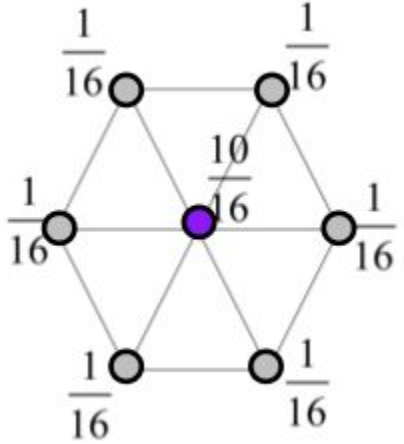
- Perturb the position of each **new vertex** (black) using a weighted average of the four nearby original vertices (grey)

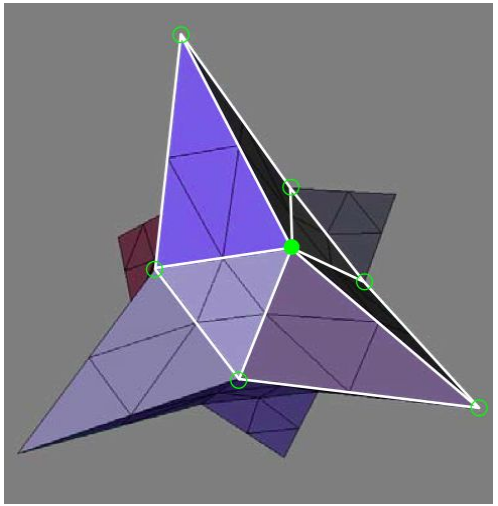


$$v_{new} = \tfrac{3}{8}(v_1 + v_2) + \tfrac{1}{8}(v_3 + v_4)$$

Loop Subdivision: Moving Old Vertices

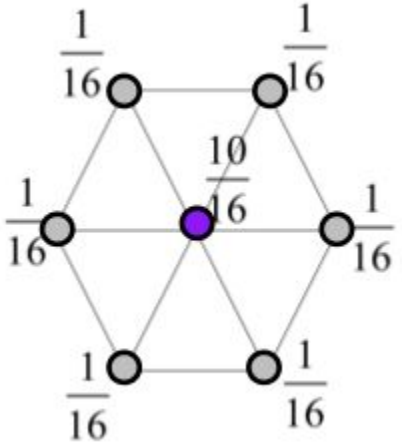
- Perturb the position of each **regular original** vertex (purple) using a weighted average of the six adjacent original vertices (grey)





Loop Subdivision: Moving Old Vertices

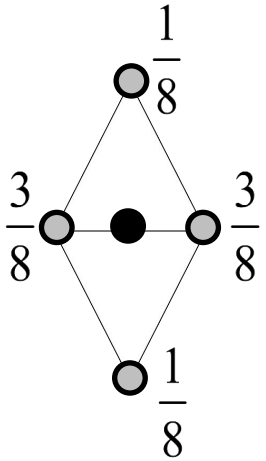
- Perturb the position of each **regular** **original** vertex (purple) using a weighted average of the six adjacent original vertices (grey)

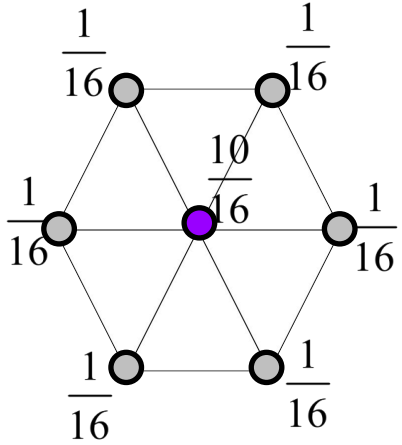


$v_{new} = 10/16\ (v_{orig}) + 1/16 \sum(i = 1,2,3,4,5,6)(v_i)$

Loop Subdivision: Moving Old/New Vertices

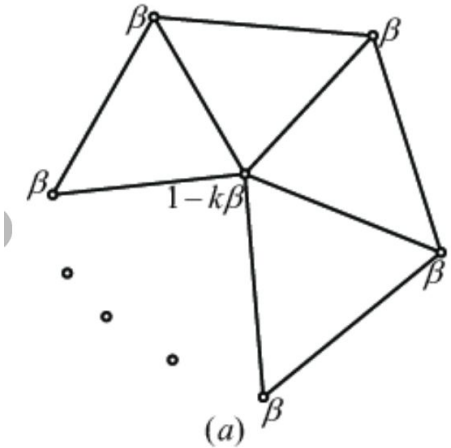
- Perturb the position of each new vertex (black) using a weighted average of the four nearby original vertices (grey)
- Perturb the position of each **regular original** vertex (purple) using a weighted average of the six adjacent original vertices (grey)



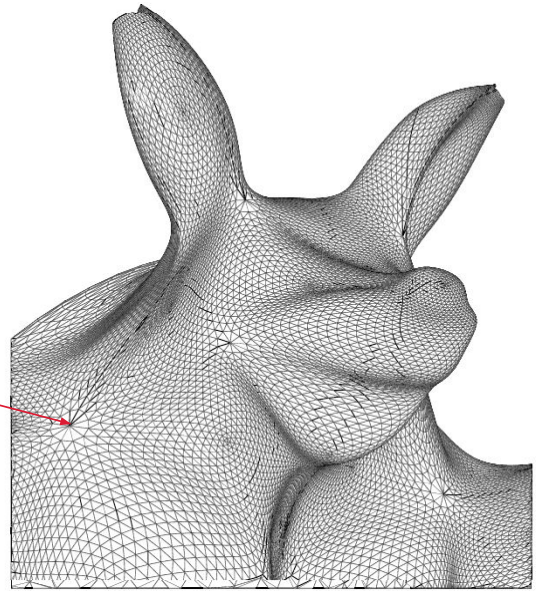


Loop Subdivision: Extraordinary Points

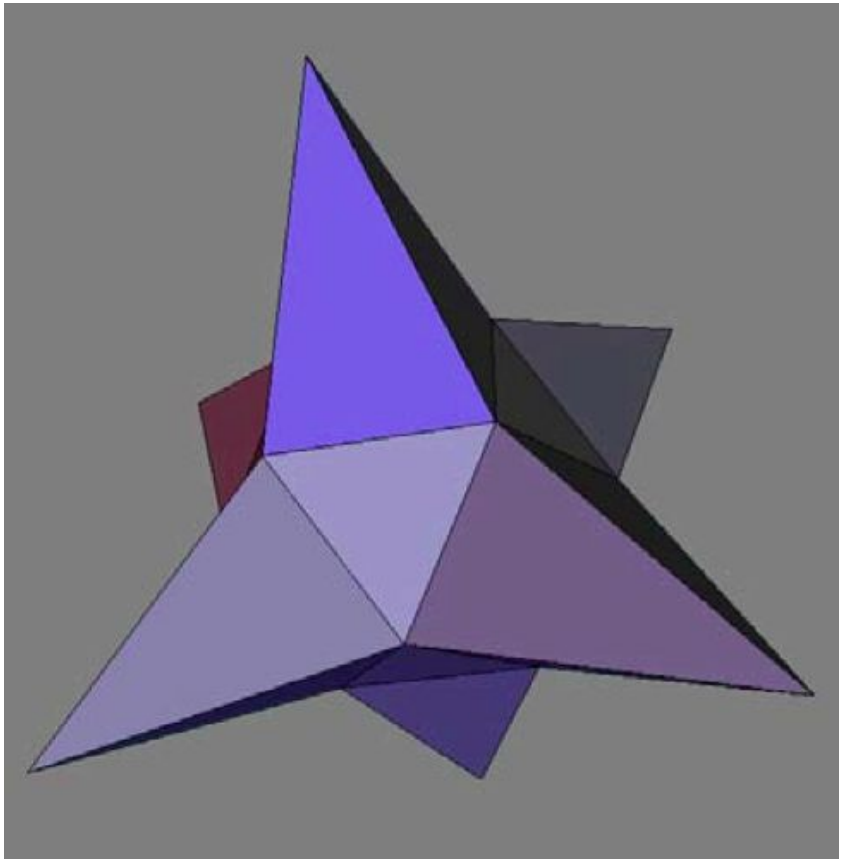
- Most vertices are regular with degree 6 (6 surrounding vertices)
- At extraordinary points, we use **Warren weights**



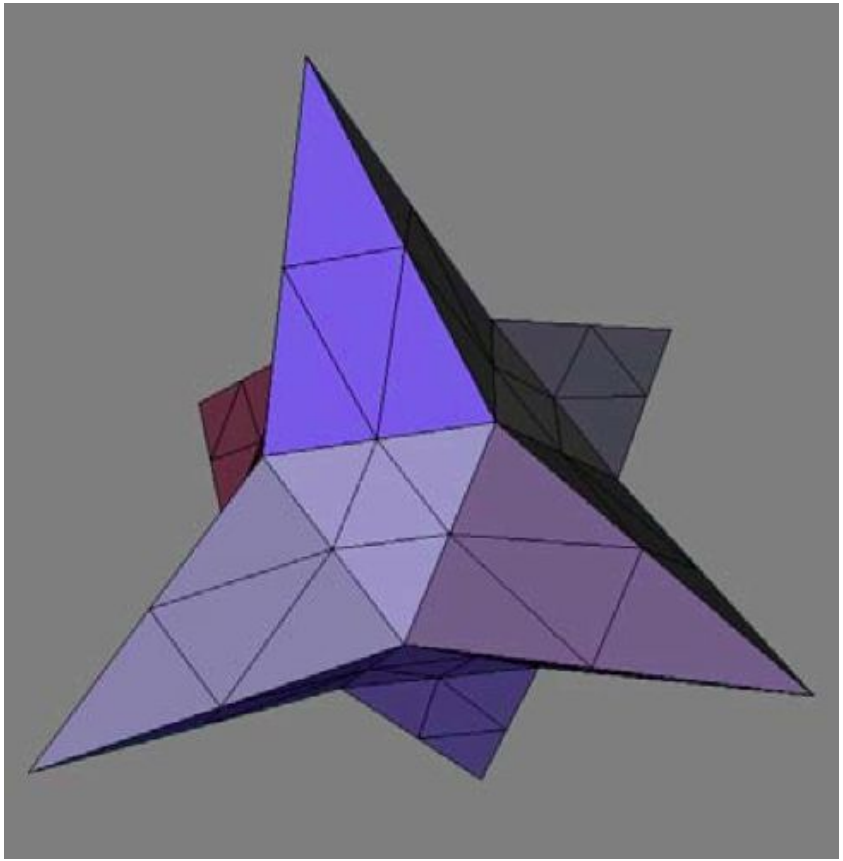
$$\beta = \frac{3}{16}, n = 3 \qquad \beta = \frac{3}{8n}, n \neq 3$$



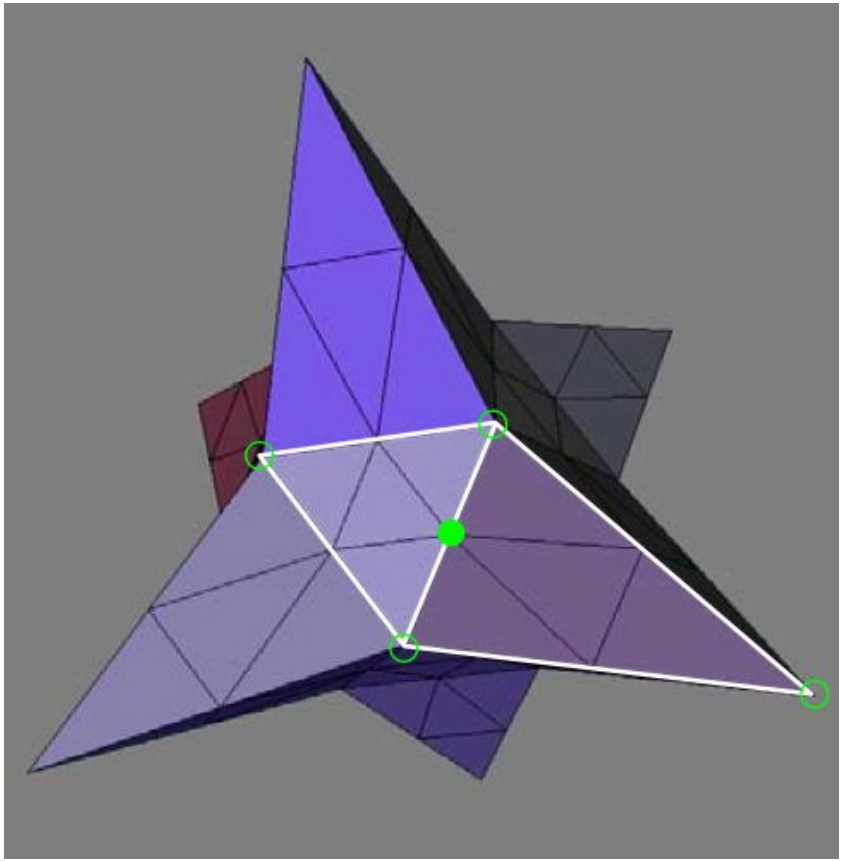
Example: Initial Mesh

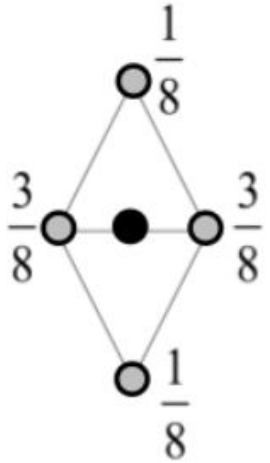


Example: Splitting, Vertices

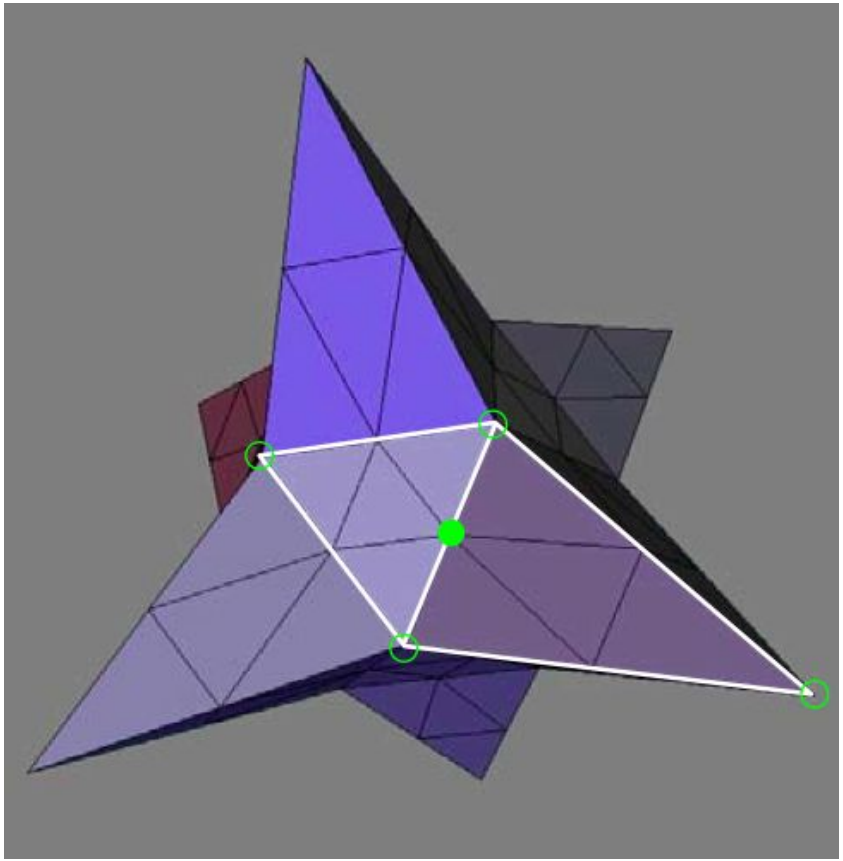


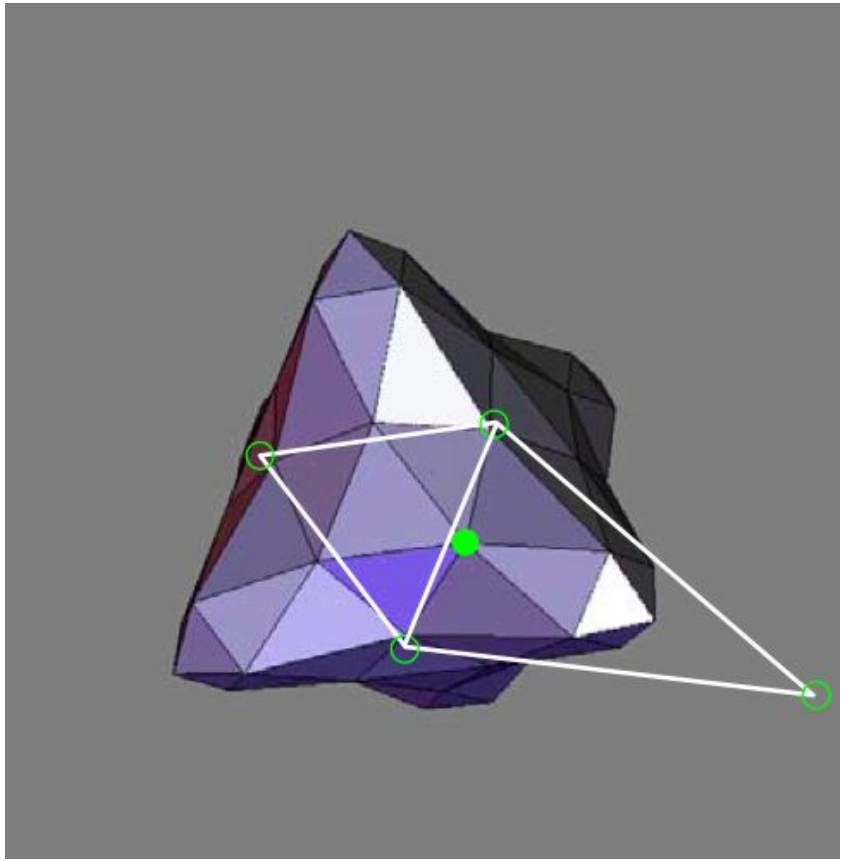
Example: Moving New Vertices



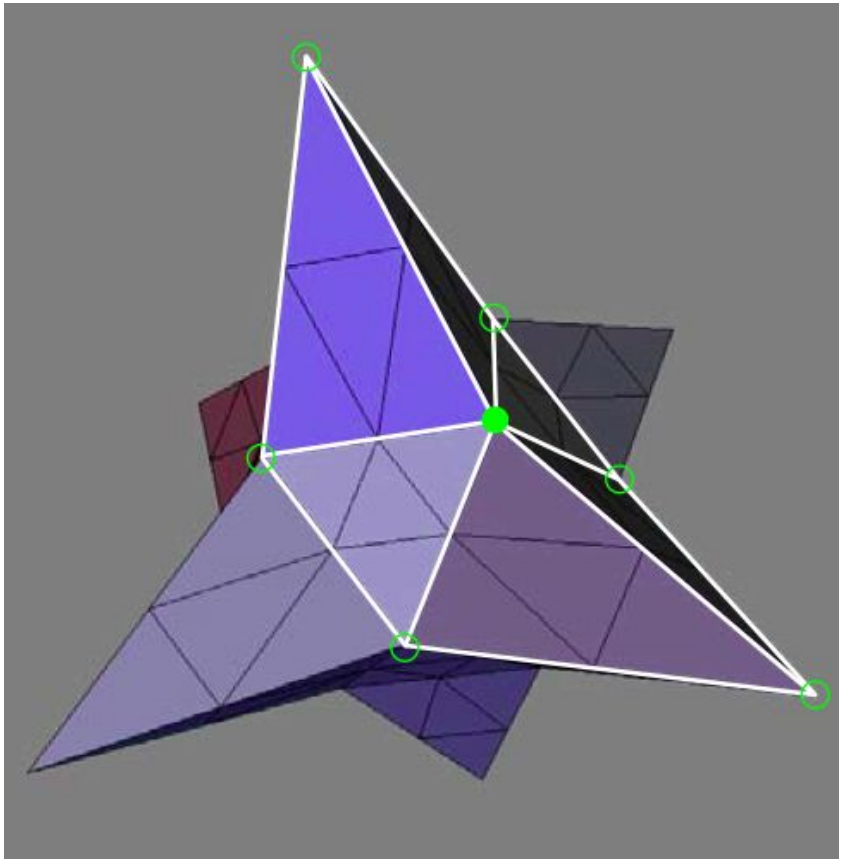


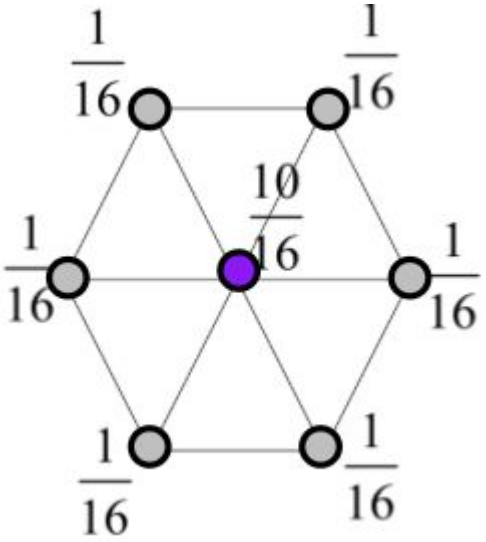
Example: Moving New Vertices



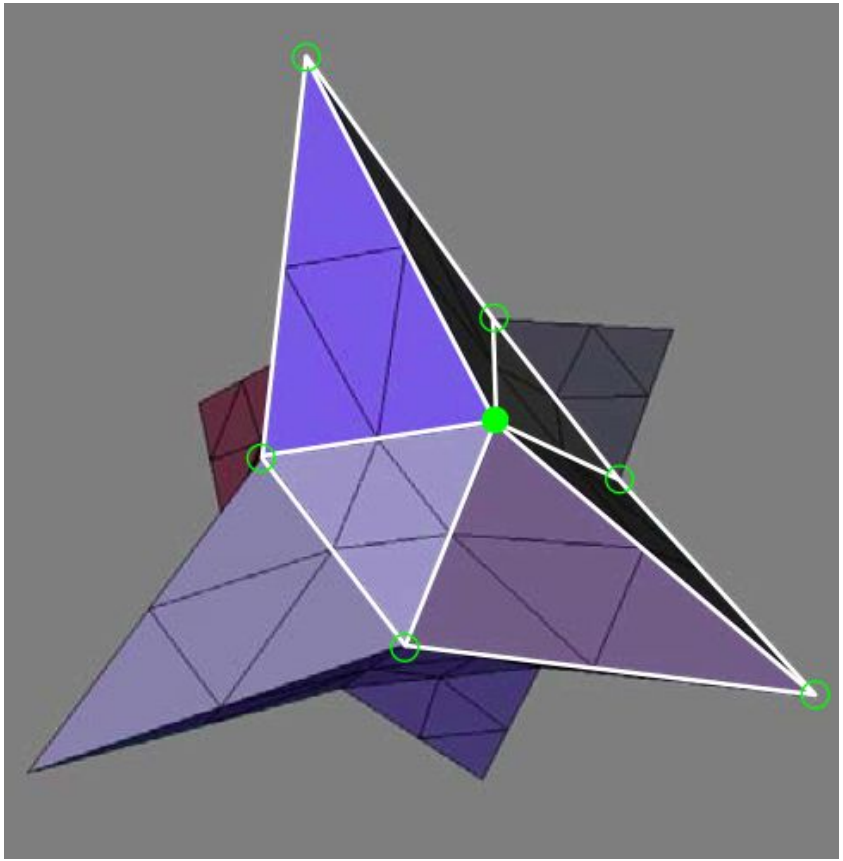


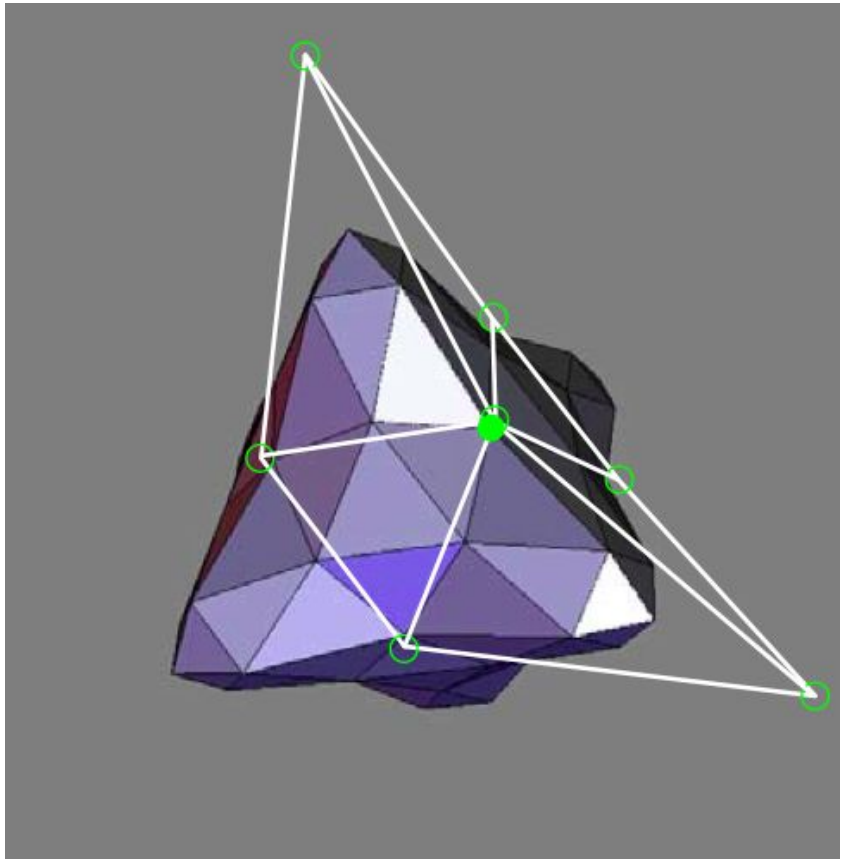
Example: Moving Old Vertices



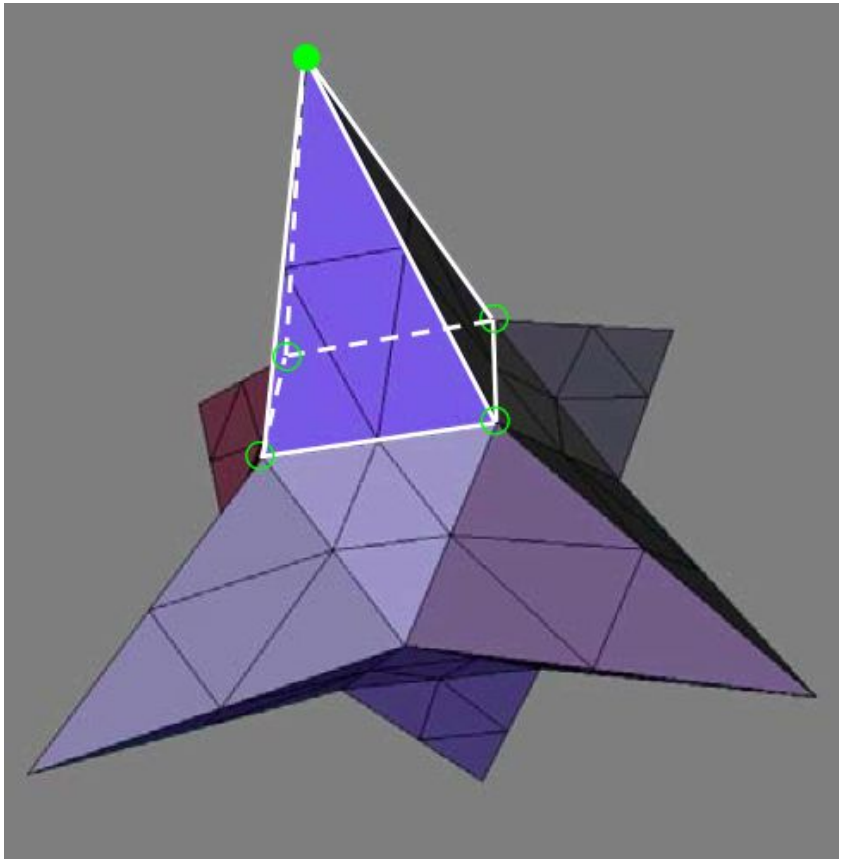


Example: Moving Old Vertices

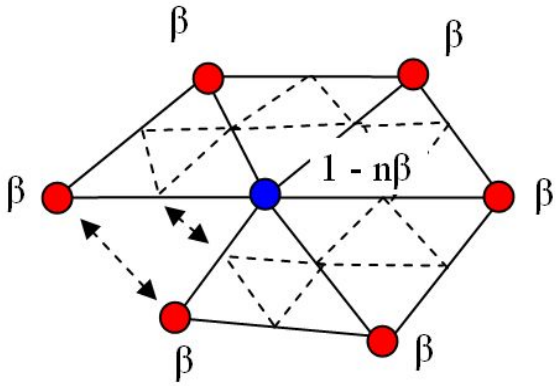




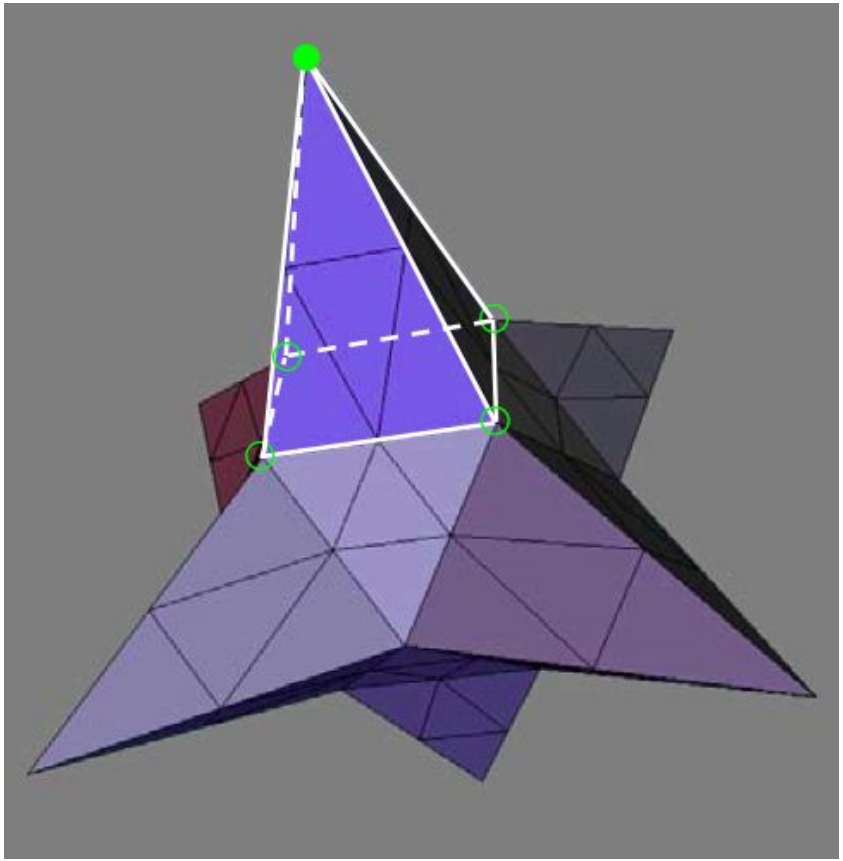
Example: Extraordinary Vertices

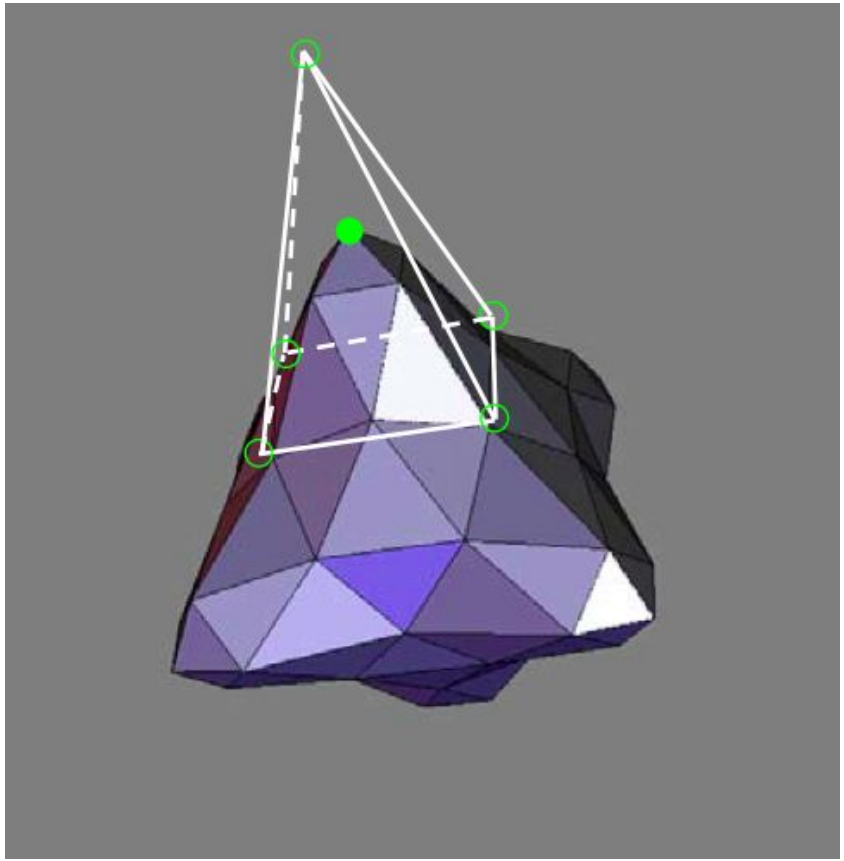


$$\beta = \frac{3}{16}, n = 3 \qquad \beta = \frac{3}{8n}, n \neq 3$$

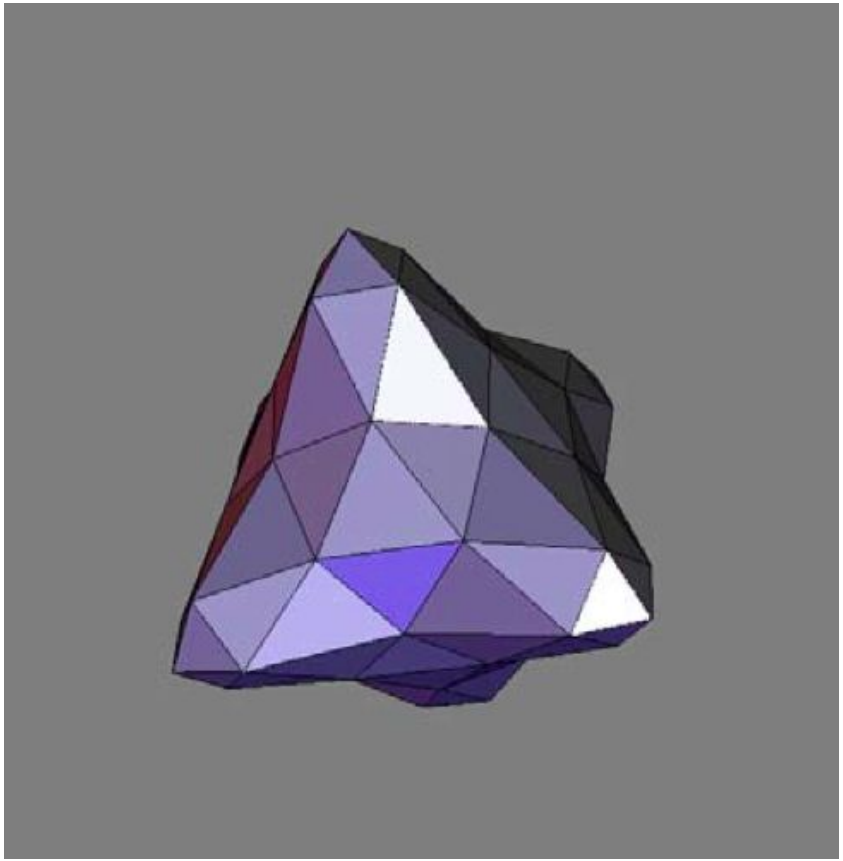


Example: Extraordinary Vertices

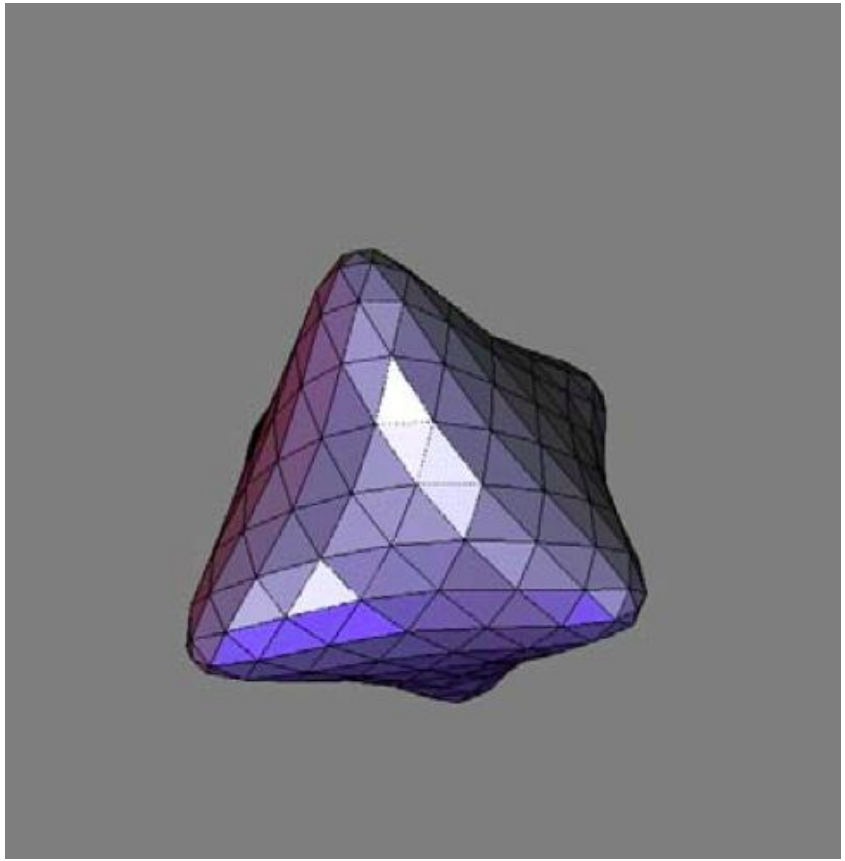




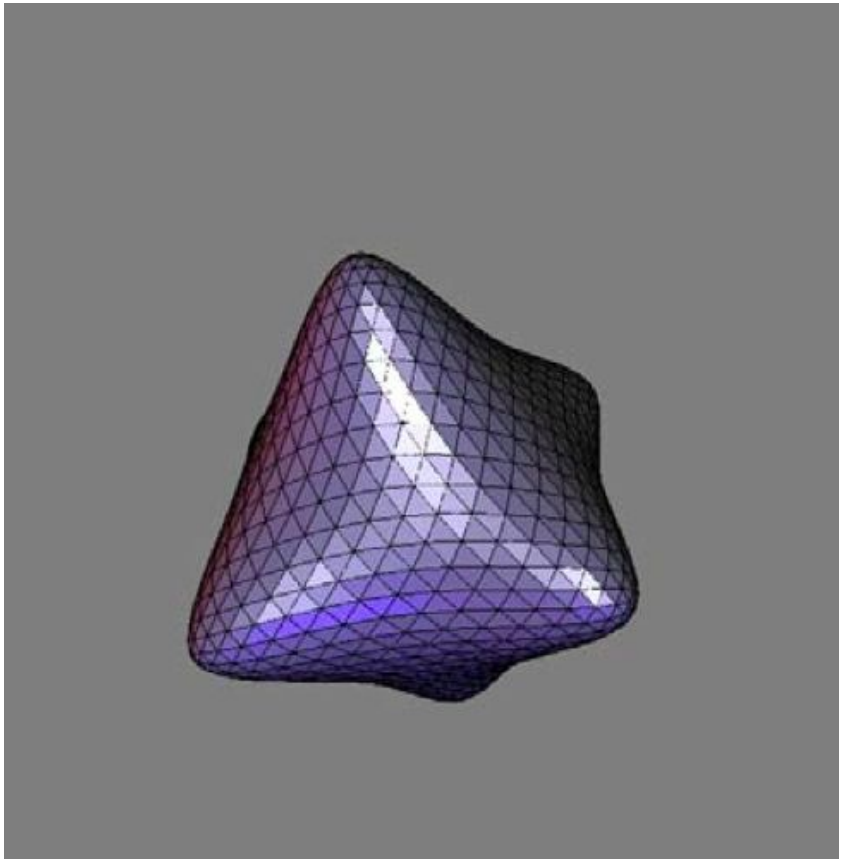
Example: Result from One Subdivision



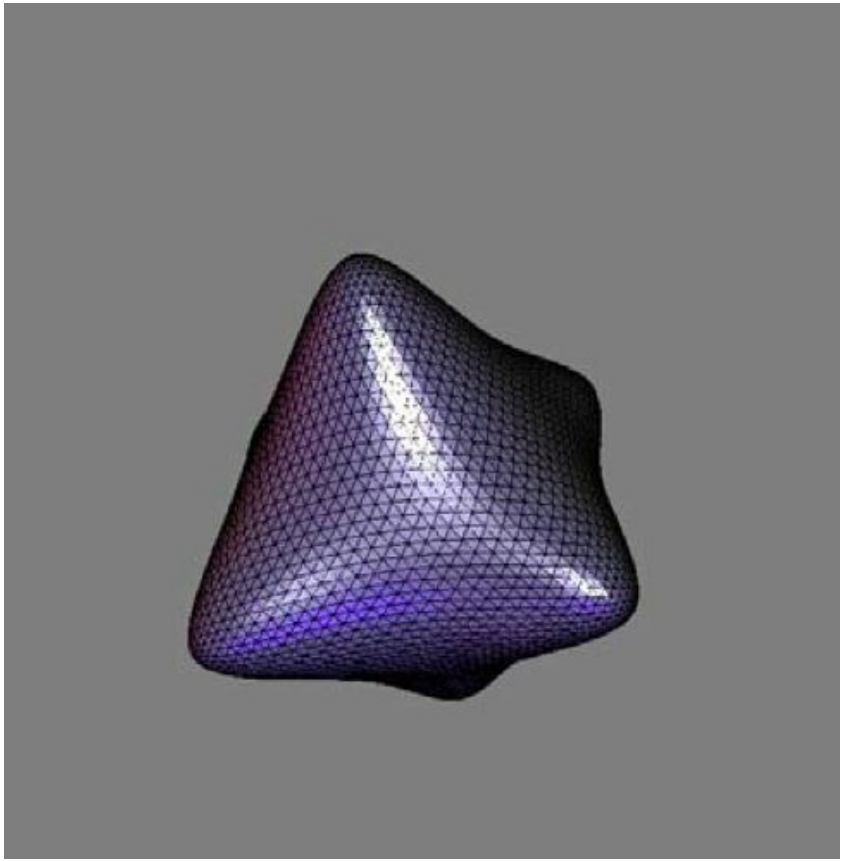
Example: Result from Two Subdivisions



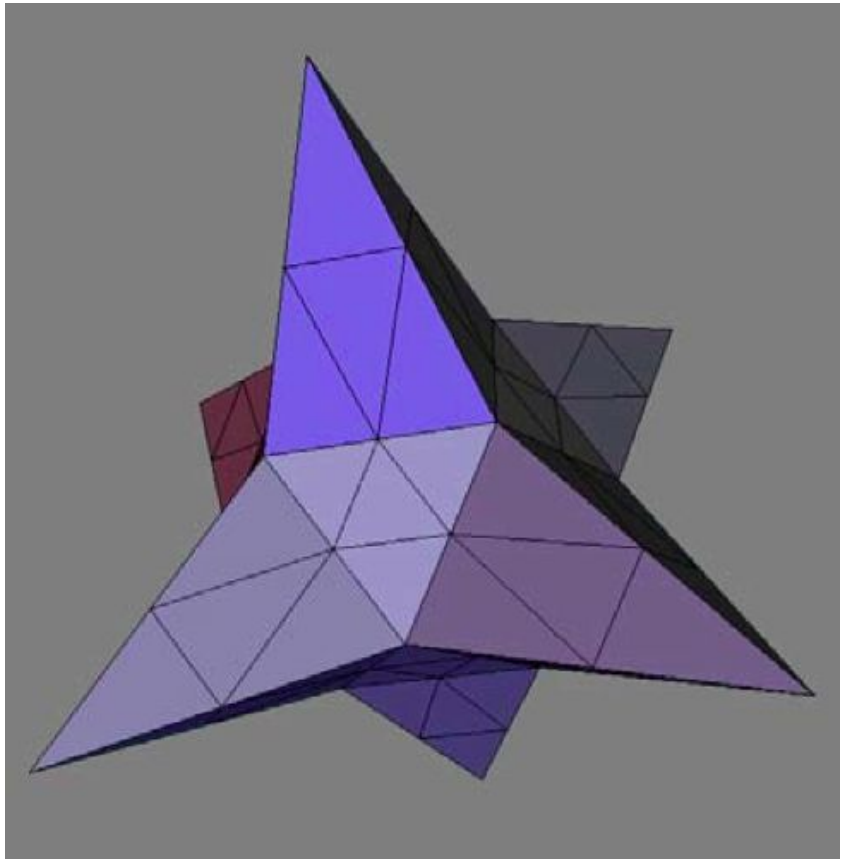
Example: Result from Three Subdivisions

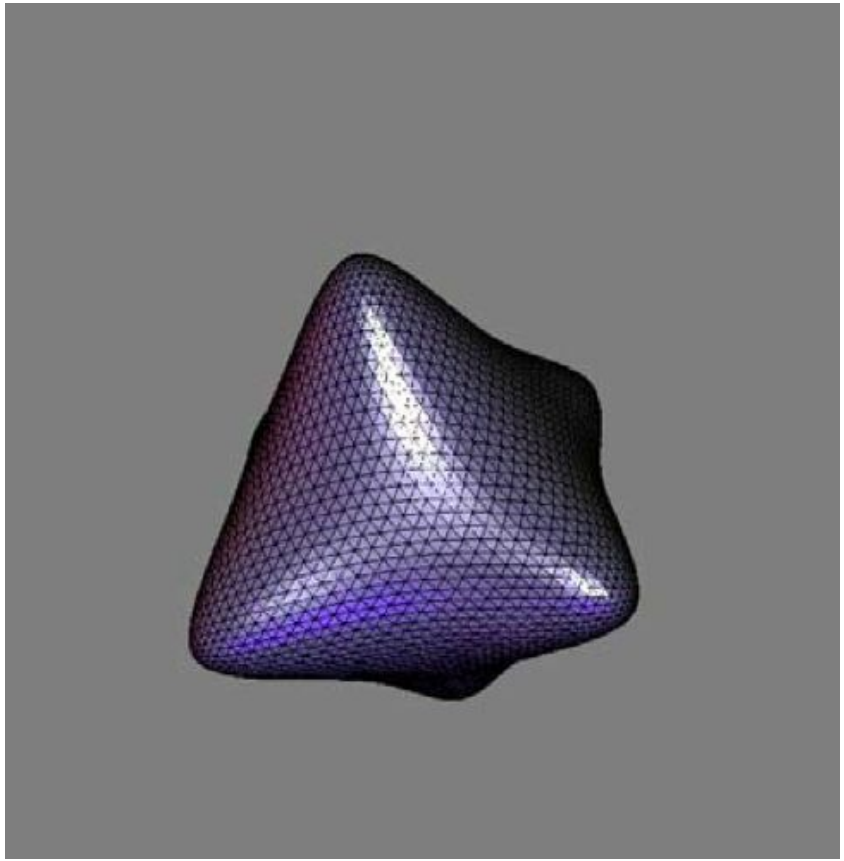


Example: Result from Four Subdivisions

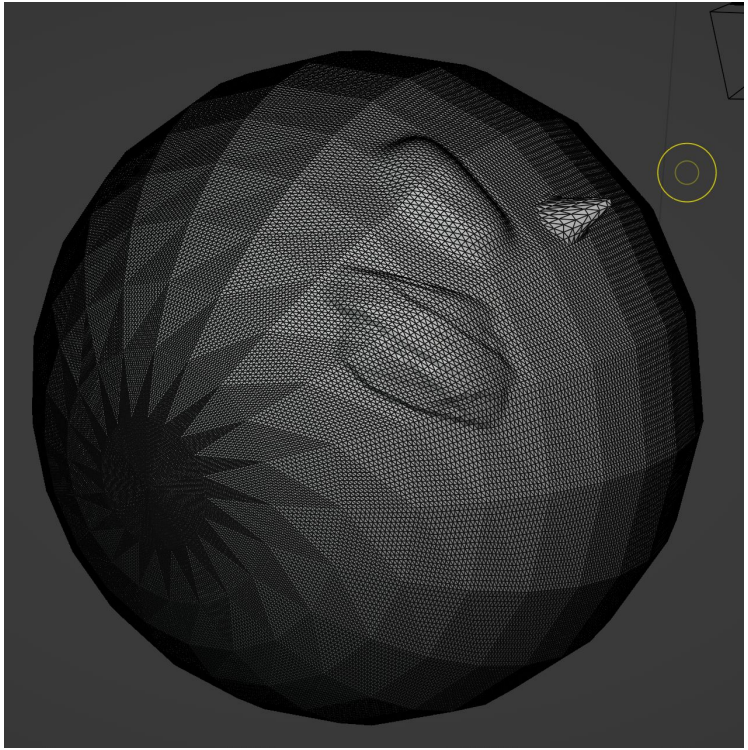


Example: Results





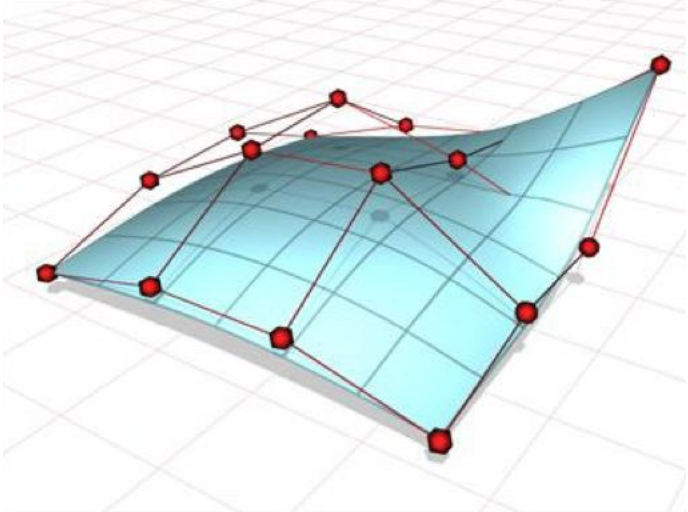
Motivation: Deforming a Mesh



Splines

- Problem: creating + moving every vertex is time intensive
- Solution: move just one vertex and have the rest move smoothly along with spline interpolation

- How Blender deforms and sculpts objects!



Linear Interpolation

- A way to estimate values between two known points
- Suppose we have some important quantities at two points p1 and p2, and we want to interpolate the quantity in between:

$$f(u) = (1 - u)p_0 + up_1$$

where our parameter (u) is between [0, 1]

Linear Interpolation

- A way to estimate values between two known points
- Suppose we have some important quantities at two points p1 and p2, and we want to interpolate the quantity in between:

$$f(u) = (1-u)p_0 + up_1$$

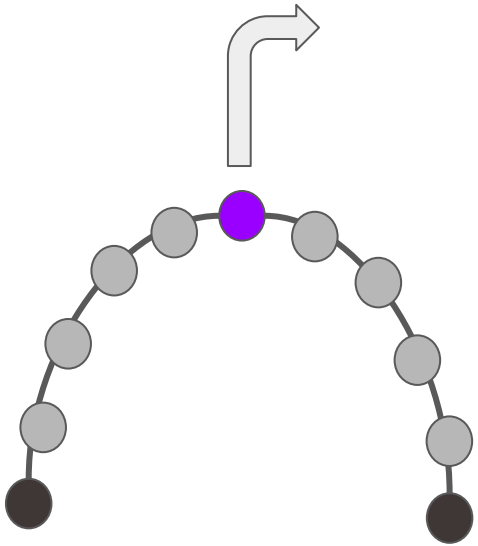
where our parameter (u) is between [0, 1]

- Expressing this as a polynomial:
where $f(0) = p_0$ and $f(u) = a_0 + ua_1$ and the a's can be solved:

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

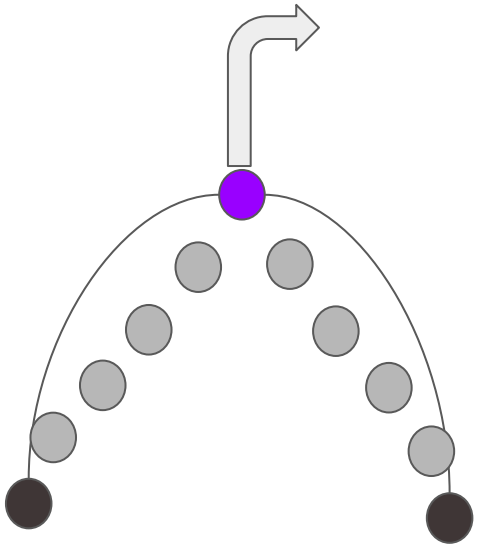
Linear Interpolation: Not Smooth Enough

Example:



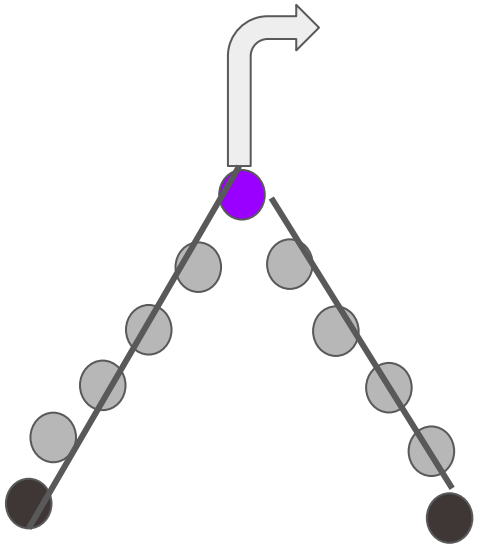
Linear Interpolation: Not Smooth Enough

Example:



Linear Interpolation: Not Smooth Enough

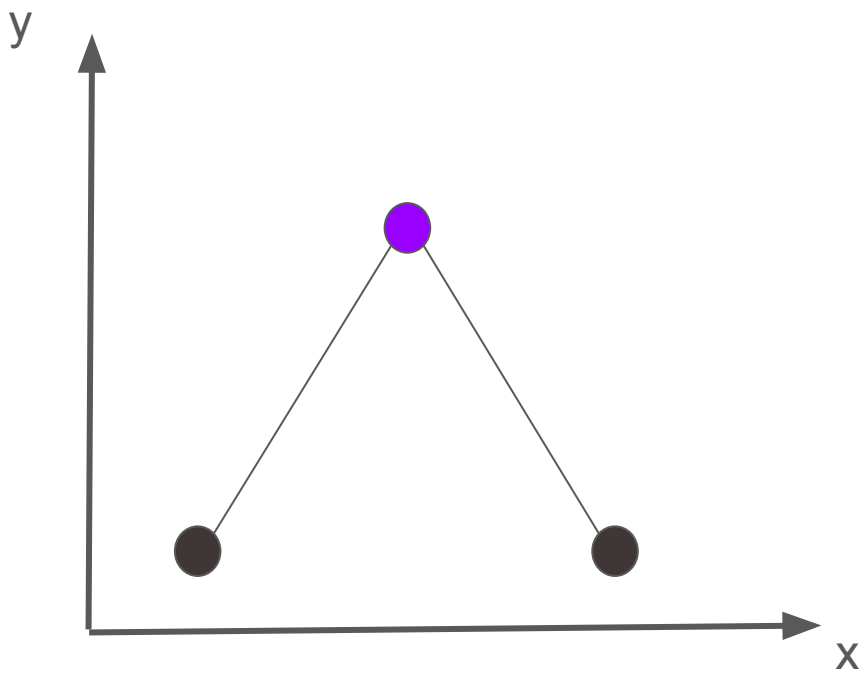
Example:



Linear Interpolation: Not Smooth Enough

Example:

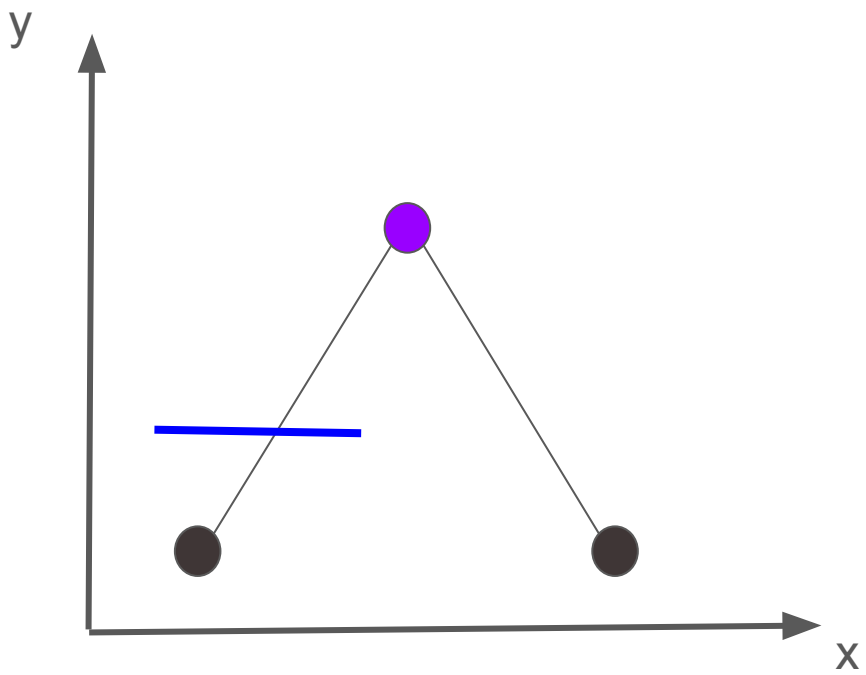
- Derivative of linear function = constant function



Linear Interpolation: Not Smooth Enough

Example:

- Derivative of linear function = constant function

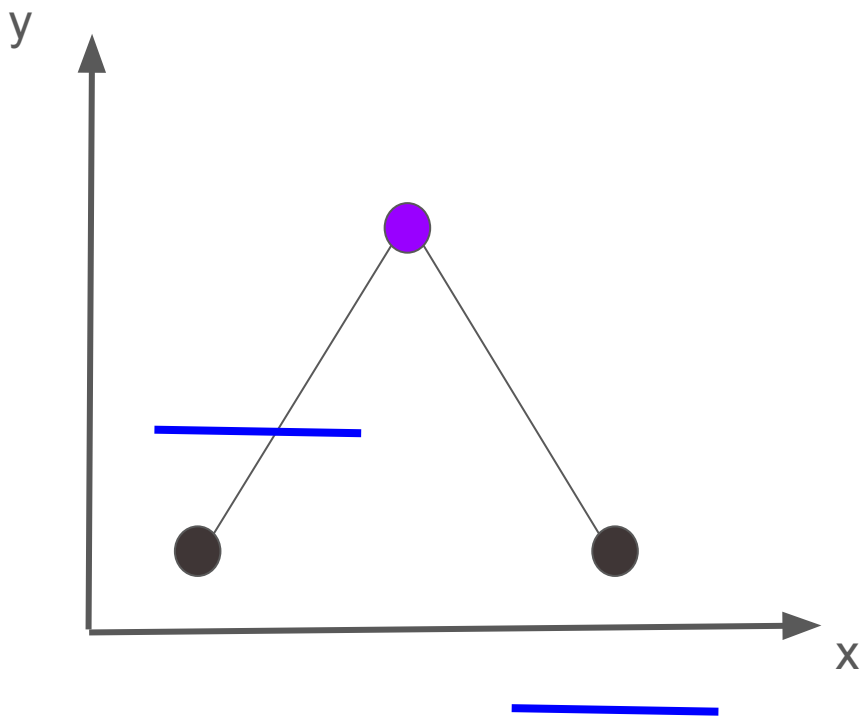


(Derivative functions in blue)

Linear Interpolation: Not Smooth Enough

Example:

- Derivative of linear function = constant function
- Discontinuous = the derivatives do not connect

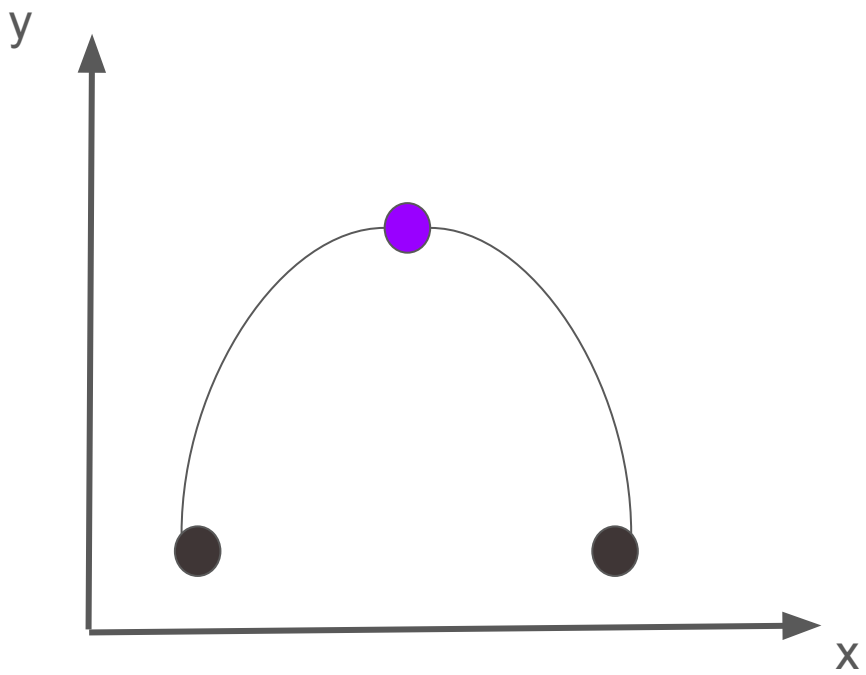


(Derivative functions in blue)

Linear Interpolation: Not Smooth Enough

Example:

- If we use quadratic…



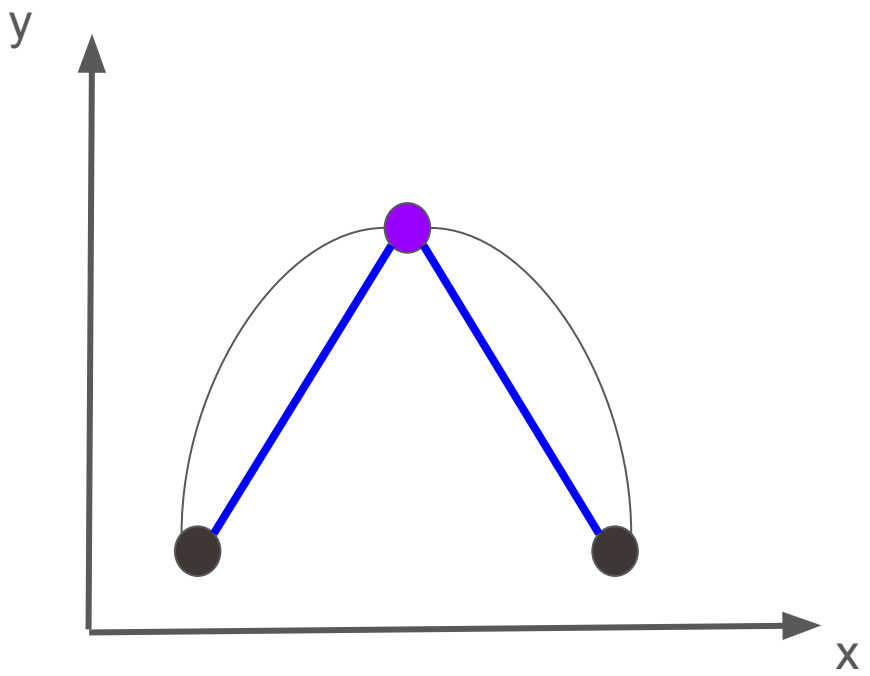
(Derivative functions in blue)

Linear Interpolation: Not Smooth Enough

Example:

- If we use quadratic…
- Derivative of quadratic function = linear function
- Derivatives are now continuous! C1 continuity
- Continuous = slope arriving at that point to equal slope leaving it

(Derivative functions in blue)



Linear Interpolation: Not Smooth Enough

- Given a bunch of control points, we want to smoothly move the rest
- But if we only do linear interpolation, then we get sharp movement
- We want at least C^1 continuity

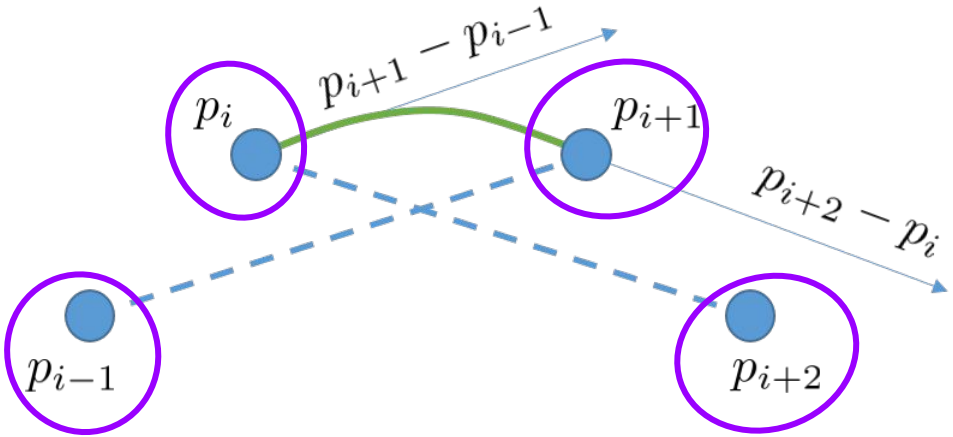
Linear Interpolation: Not Smooth Enough

- Given a bunch of control points, we want to smoothly move the rest
- But if we only do linear interpolation, then we get sharp movement
- We want at least C^1 continuity

- Turns out, cubics are enough for most applications, with C^1 & C^2 continuity
- Many kinds of cubic splines

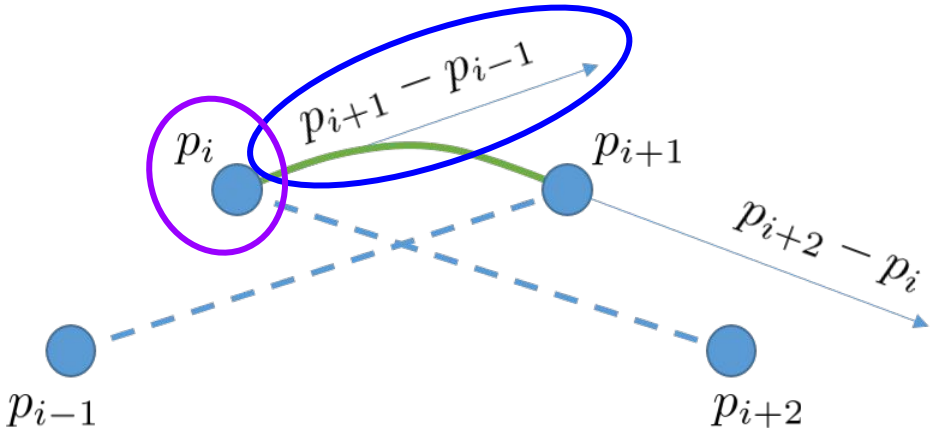
Cardinal Cubic Splines

- 4 control points



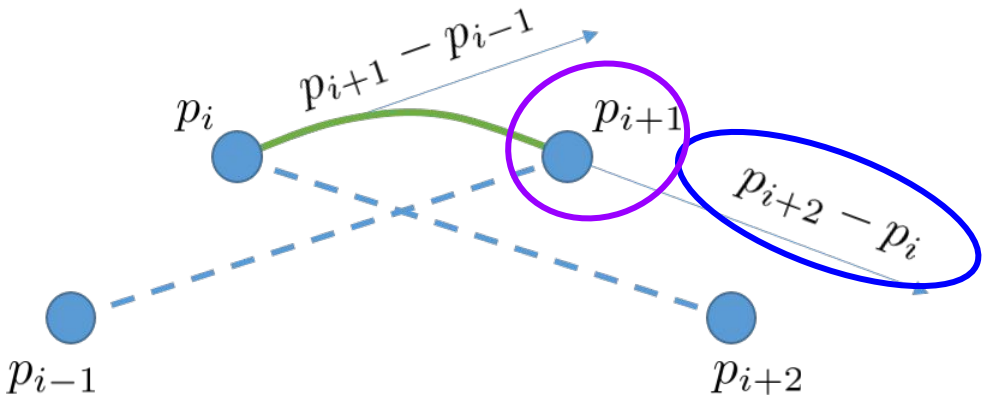
Cardinal Cubic Splines

- 4 control points
- Derivative at the 2nd control point connects the 1st & 3rd



Cardinal Cubic Splines

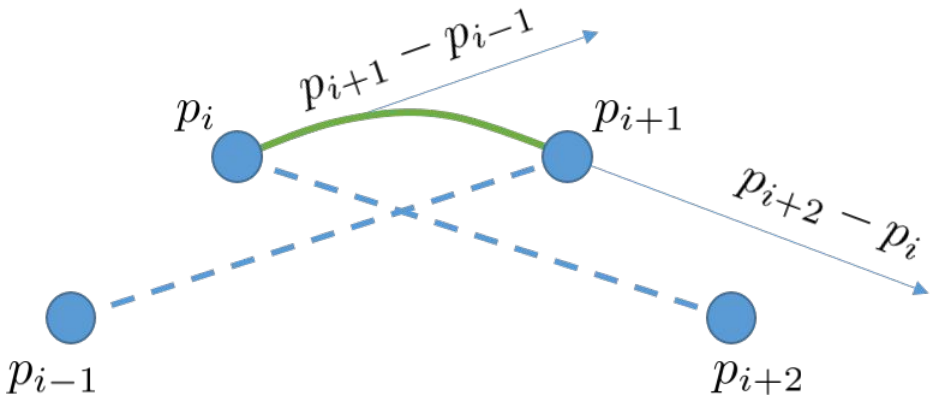
- 4 control points
- Derivative at the 2nd control point connects the 1st & 3rd
- Derivative at the 3rd control point connects the 2nd & 4th



Cardinal Cubic Splines

- 4 control points
- Derivative at the 2nd control point connects the 1st & 3rd
- Derivative at the 3rd control point connects the 2nd & 4th

- Construction of derivatives makes 2 consecutive curves continuous!

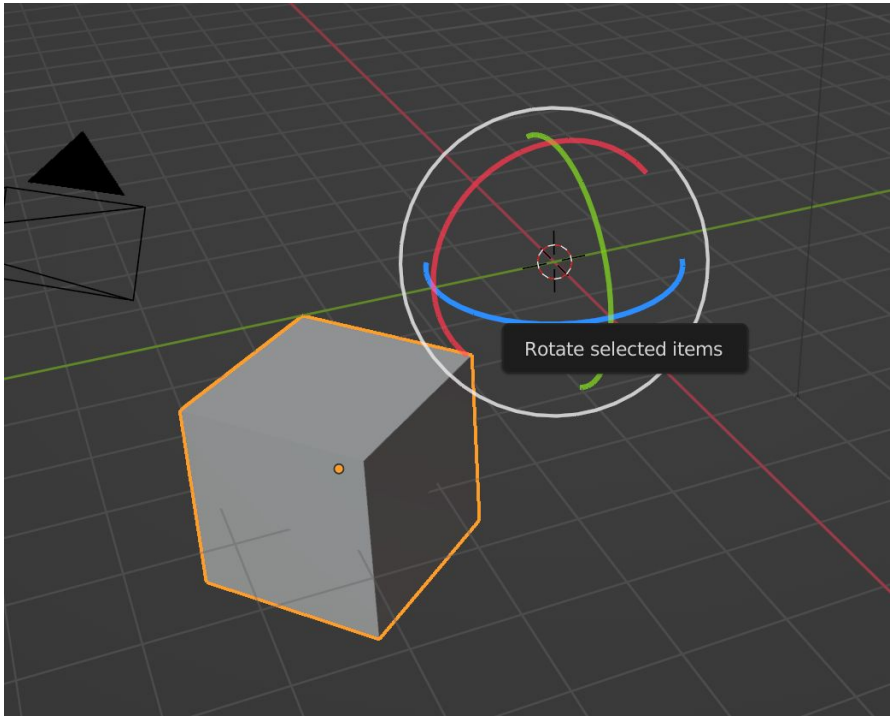


Lecture Outline

- Points and the GPU
- Triangles
- Subdivision
- Transformations

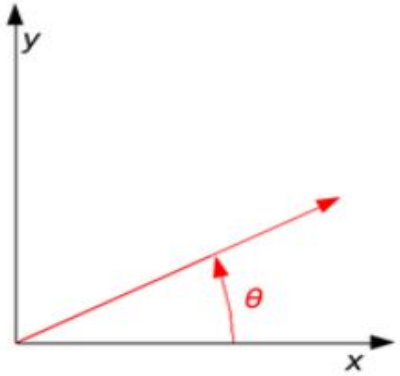
Basics

- We create objects using global coordinates
- After creating them, we place the objects into the scene (aka world space)
- Placing objects includes rotation, scaling (resizing), or translating



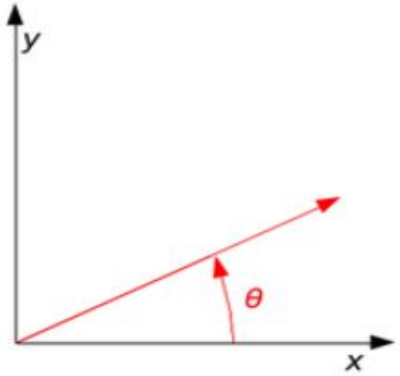
Rotation

- In 2D, we rotate a point counter-clockwise about the origin of a Cartesian coordinate space



Rotation

- In 2D, we rotate a point counter-clockwise about the origin of a Cartesian coordinate space
- We use the rotation matrix to do so (boxed in blue)



$$\begin{pmatrix} x^{new} \\ y^{new} \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = R(\theta) \begin{pmatrix} x \\ y \end{pmatrix}$$

Rotation

- In 3D, there are rotation matrices for about the x, y, z axes respectively:

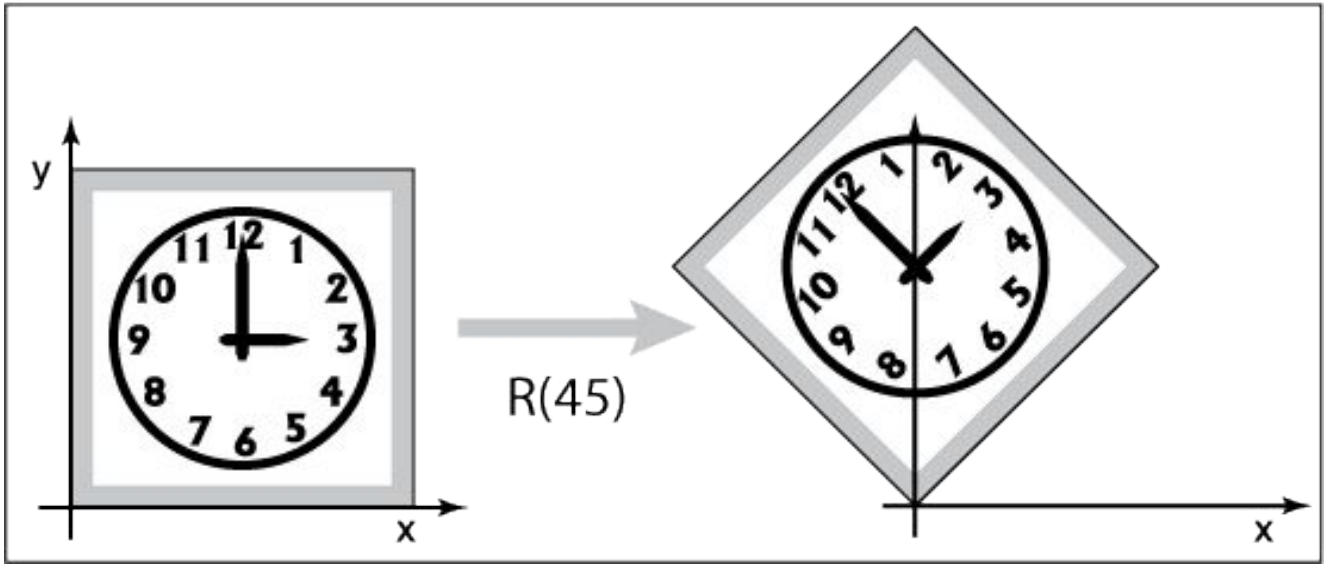
$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotation

- Rotations preserve shape
- Order of rotation matters
  - Matrix multiplication is not commutative

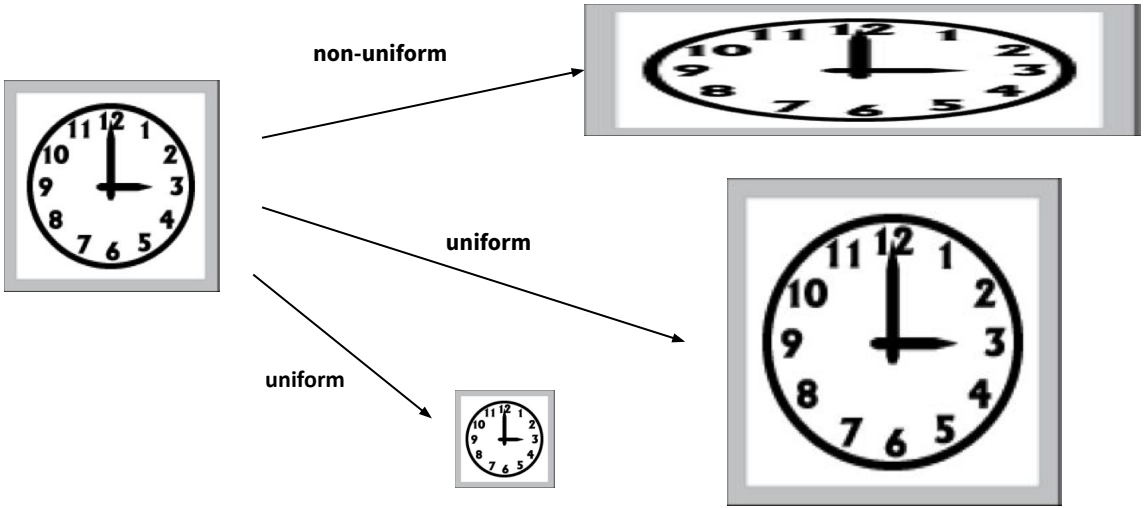


Scaling (aka Resizing)

- A scaling matrix has the form:

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Scaling matrices can both resize and shear/stretch objects:



Homogeneous Coordinates

- To represent translation with matrices, we need to use homogeneous coordinates

- In general, the homogeneous coordinates of a point in 3D are:

$$\vec{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \vec{p}_H = \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}, w \neq 0$$

- Let the 4th component be 1, so we have: $\vec{p}_H = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

Translation

- To translate a point some amount $\vec{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$ we use the 4x4 matrix:

$$\begin{pmatrix} & I_{3x3} & & t_1 \\ & & & t_2 \\ & & & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \vec{x} + \vec{t} \\ 1 \end{pmatrix}$$

with the identity: $I_{3x3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Translation

- To translate a point some amount $\vec{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$ we use the 4x4 matrix:

$$\begin{pmatrix} & & & t_1 \\ & I_{3x3} & & t_2 \\ & & & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \vec{x} + \vec{t} \\ 1 \end{pmatrix}$$

with the identity: $I_{3x3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

- Intuitively, this should confirm with what you'd expect with translation
  - Moving a point by some vector simply adds that vector to the point to get the new location

Transforming in Homogeneous Coordinates

- Rotation and scaling matrices can also be expressed in homogeneous coordinates as:

$$\begin{pmatrix} & M_{3x3} & & \begin{matrix}0\\0\\0\end{matrix} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} M_{3x3}\vec{x} \\ 1 \end{pmatrix}$$

Transforming in Homogeneous Coordinates

- Rotation and scaling matrices can also be expressed in homogeneous coordinates as:

$$\begin{pmatrix} & M_{3x3} & \\ & & \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} M_{3x3}\vec{x} \\ 1 \end{pmatrix}$$

- Notice how we can simply take off the fourth component, the 1, once we're done with our transformations and get our desired transformed vertex

Transforming in Homogeneous Coordinates

- In short, to transform a 3D object:

Transforming in Homogeneous Coordinates

- In short, to transform a 3D object:
  - Convert all vertices in the object to homogeneous coordinates by simply making them 4D vectors with a fourth 1 component

Transforming in Homogeneous Coordinates

- In short, to transform a 3D object:
  - Convert all vertices in the object to homogeneous coordinates by simply making them 4D vectors with a fourth 1 component
  - Apply your transformation matrices for translation, rotation, scaling by left-multiplying each vertex in the order you want to apply the transforms
    - `final_point = T · (R · (S · initial_point))`
    - ^ means that scale happens first, then rotation, then translation

Transforming in Homogeneous Coordinates

- In short, to transform a 3D object:
  - Convert all vertices in the object to homogeneous coordinates by simply making them 4D vectors with a fourth 1 component
  - Apply your transformation matrices for translation, rotation, scaling by left-multiplying each vertex in the order you want to apply the transforms
    - `final_point = T · (R · (S · initial point))`
    - ^ means that scale happens first, then rotation, then translation
  - Convert all vertices back to 3D by taking off the fourth component once you're done

Additional Resources + Exploring!

- History of the OBJ File Format
- The Digital Michelangelo Project
- Why do 3D engines use triangles to draw object surfaces?
- Spherical Geometry
- Cubic Spline Interpolation

- Fundamentals of Computer Graphics, Steve Marschner and Peter Shirley
  - Preview, Book