Computer Graphics

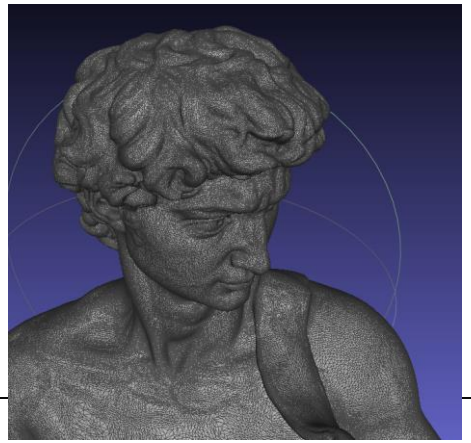
Prof. Jibum Kim

Department of Computer Science & Engineering

Incheon National University

- **Level of detail (LOD) and meshes**

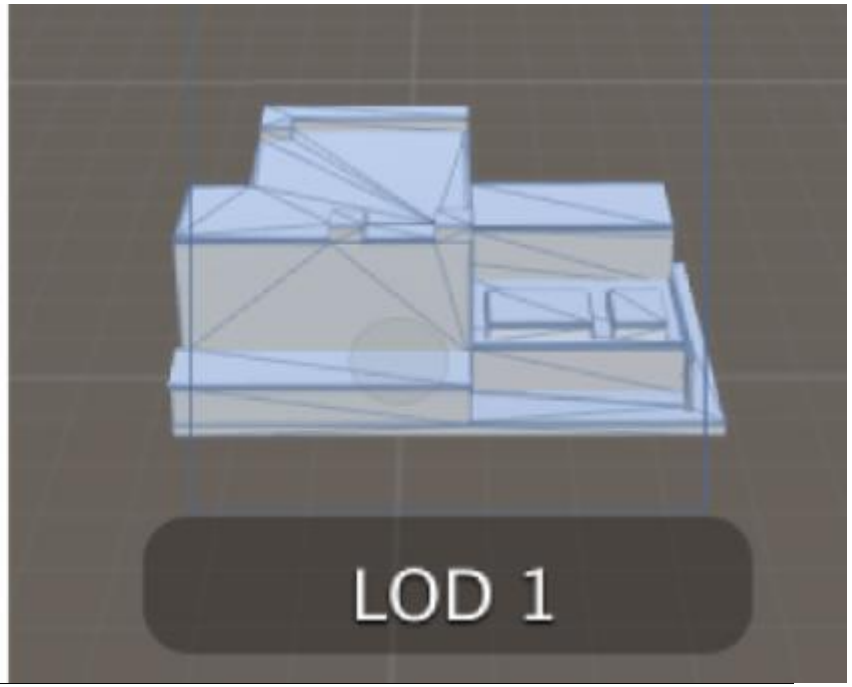
- **Very large (mesh) models**
- **Michelangelo's David: 56 million polygons**
- **More polygons in your scene**
- **More detailed objercts, visually richer**
- **But it requires fast hardware rendering speeds**
- **David (Michelangelo) - File:David (Michelangelo).stl - Wikimedia Commons**



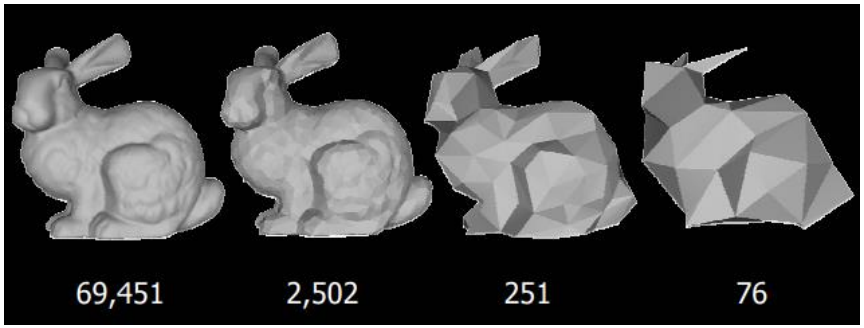
- **Level of detail (LOD)**
- **Allow objects to be represented with different number of polygons**
- **Use fewer polygons for distant objects**
  - Less visual contribution
- **Use more polygons for near objects**
  - More visual contribution

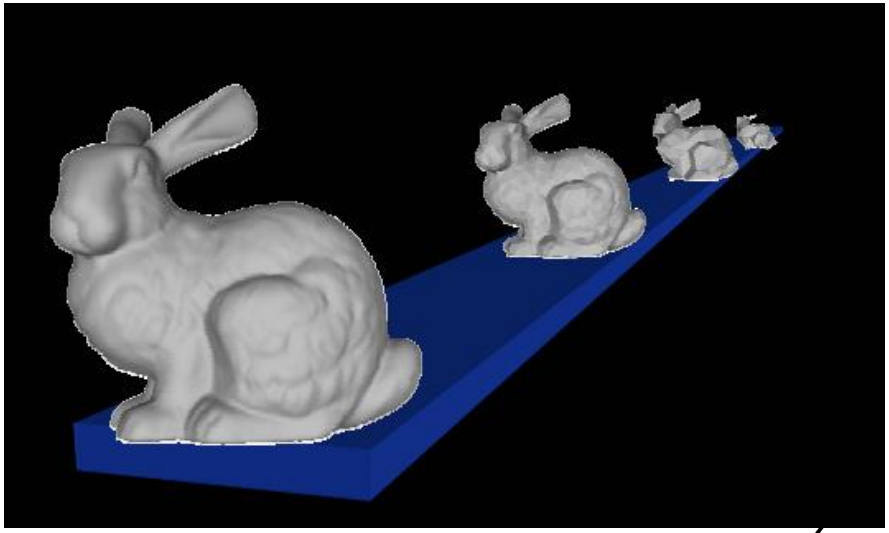
- **Unity LOD meshes**
- **메시용 디테일 수준(LOD) - Unity 매뉴얼 (unity3d.com)**
- 디테일 수준(LOD)은 Unity가 멀리 있는 메시를 렌더링하는 데 필요한 GPU 연산 수를 줄여주는 기술입니다.
- 게임 오브젝트가 카메라에서 멀리 떨어질수록 Unity는 더 낮은 디테일의 LOD 레벨을 렌더링합니다. 이 기술은 멀리 떨어져 있는 게임 오브젝트에 대한 하드웨어 부하를 줄이므로 렌더링 성능이 향상될 수 있습니다.

- **Unity LOD level**
- **카메라에서 떨어진 거리를 기반으로 detail 수준 (LOD level)을 정함**



- **Bunny example**
- **Stanford bunny models**
- **The Stanford 3D Scanning Repository**

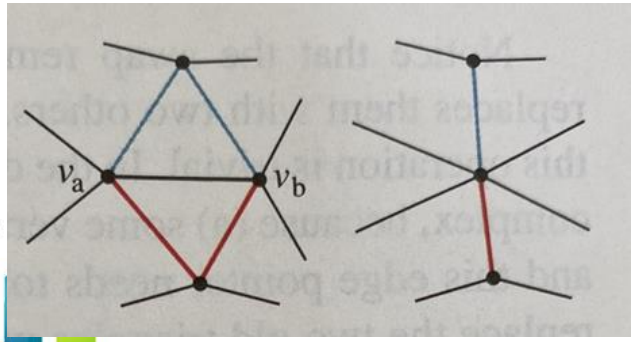


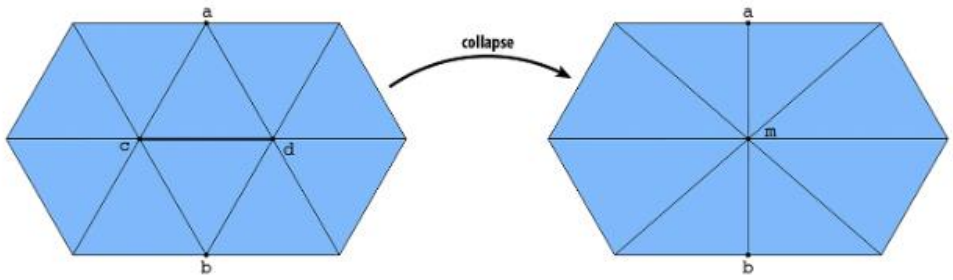


- **LOD generation and mesh simplification**

- **Mesh simplification: a mesh is replaced by another mesh that is similar to it but has a more compact structure**
- **LOD에서 꼭 필요한 mesh operation**

- One of the standard operation for mesh simplification is an edge collapse, in which one edge is shrunk until it has length zero, resulting in the two adjacent triangles disappearing
- The edge from vertex a to vertex b is collapsed; the edge itself and the two adjacent faces are removed from the data structure; and the other two edges of the upper face, drawn in blue, becoming one, as do the two edges of the lower face, drawn in red
- The two vertices va and vb become a single vertex

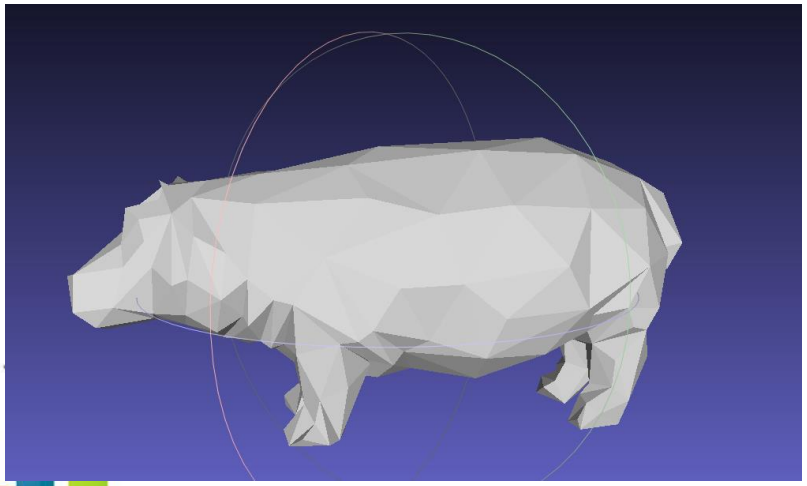


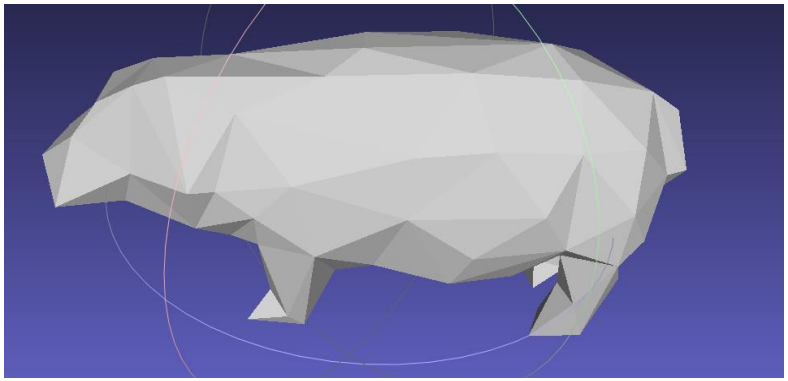


- There is a geometric question as well: when we merge the two vertices, we must choose a location for the merged vertex
- The location we choose depends on the goal of our simplification
- 1. computation is important: simply use one of the old vertices
- 2. If we want to preserve some sort of shape, averaging the two vertices is easy
- There is no one “right answer”

- **Edge collapse and Meshlab**

- **Meshlab에서**
- **File -> import Mesh -> Filters -> Remeshing, Simplification and Reconstruction -> Simplification: Quadric Edge Collapse Decimation**
- **다양한 option들 있음. Targe number of faces 정함**



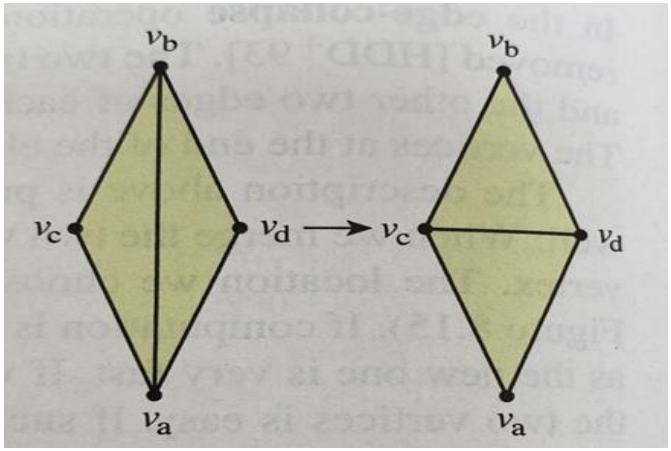


- **Edge swap**

- **High aspect ratio (long and thin) triangles produce bad artifacts in many situations, so it is nice to be able to eliminate them when possible**

- **An edge-swap (flip) operation can convert two adjacent high-aspect-ratio triangles to two with low aspect ratios**

- **In this example, the triangles adjacent to the edge from va to vb both have bad aspect ratios**
- **By replacing the edge $v_a v_b$ with the edge $v_c v_d$, we get a new pair of triangles**



- **Opacity and blending**

- We have assumed that we want to form a single image and that the objects that form this image have surfaces that are opaque
- So far we have used RGBA colors and used an alpha (α) of 1
- OpenGL provides a mechanism, through alpha blending, that can create images with translucent objects
- The alpha (α) channel is the fourth color in RGBA color mode
- Like the other colors, the application program can control the value of α for each pixel
- However, in RGBA mode, if blending is enabled, the value of α controls how the RGB values are written into the framebuffer

- **The opacity of a surface is a measure of how much light penetrates through that surface**
- **An opacity of 1 (α=1) corresponds to a completely opaque surface that blocks all light incident on it**
- **A surface with an opacity of 0 is transparent; all light passes through it**
- **The transparency or translucency of a surface with opacity α is given by 1- α**

- A rasterized pixel, together with its color values and z-value is called a fragment
- Until now, we have used the color of a fragment to determine the color of the pixel in the framebuffer in screen coordinates of the fragment
- If we regard the incoming fragment as the source pixel and the framebuffer pixel as the destination (이미 framebuffer에 있는), we can combine these values in various ways
- Using α values is one way of controlling the blending on a fragment-by-fragment basis

- 가장 기본 적인 blending 수식
- Destination과 source의 색이 섞는 비율을 blending factor라 한다
- Source pixel (s)=$[\ s_r \quad s_g \quad s_b \quad s_a]$
- Destination pixel (d)=$[\ d_r \quad d_g \quad d_b \quad d_a]$
- Source blending factor (b)=$[\ b_r \quad b_g \quad b_b \quad b_a]$
- Destination blending factor (c)=$[\ c_r \quad c_g \quad c_b \quad c_a]$
- Blending replaces d with d'
- d'=$[b_r s_r + c_r d_r \quad b_g s_g + c_g d_g \quad b_b s_b + c_b d_b \quad b_a s_a + c_a d_a]$

- **Source의 alpha값을 이용한 blending**

- **Source의 alpha값을 이용한 blending 수식**
- **Blending factor를 정의하는 방법은 많지만 흔히 사용되는 방법은 source의 alpha값을 이용하여 source와 destination의 blending factor를 정한다**
- **Source: incoming fragment의 색 (R,G,B)=(src_R, src_G, src_B)**
- **Destination: 현재 이미 저장된 frame buffer의 색 (R,G,B)=(dst_R, dst_G, dst_B)**
- **Source의 blending factor: src_α (source의 alpha값으로 [0,1] 사이 값)**
- **Destination의 blending factor: 1-src_α**

최종색 = src_α *(source의 색) + (1- src_α) * (Destination의 색)

$$(R_f, G_f, B_f) = src_\alpha * (src_R, src_G, src_B) + (1 - src_\alpha) * (dst_R, dst_G, dst_B)$$

- 예: Consider a pixel. Say its RGB color values prior to the rendering is (0.6, 0.4, 0.2) and the RGB color values of newly rasterized one is (0.5, 0.5, 0.5). Suppose the blending factors of the source and destinations are (src_R, src_G, src_B)=(0.3, 0.3, 0.3) and (dst_R, dst_G, dst_B)=(0.7, 0.7, 0.7). What are the final color values of the pixel when blending is enabled?
- Source color? (0.5, 0.5, 0.5)
- Destination color? (0.6, 0.4, 0.2)
- $(R_f, G_f, B_f) = src_\alpha * (src_R, src_G, src_B) + (1 - src_\alpha) * (dst_R, dst_G, dst_B)$
- =(0.3*0.5+0.7*0.6, 0.3*0.5+0.7*0.4, 0.3*0.5 + 0.7*0.2)
- =(0.57, 0.43, 0.29)

- **OpenGL에서의 blending 사용**

1. OpenGL에서 Blending enable 시킴

- glEnable(GL_BLEND);

2. Blending시 source와 destination의 blending factor 정함

glBlendFunc(source의 blending factor, destination의 blending factor)

예: source의 blending factor: src_α

destination의 blending factor: $1 - src_\alpha$

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- 예: 투명도 및 blending 에 대한 코드
- 무엇이 source이고 무엇이 destination인지 파악해 보자
- 이 경우에는 destination이 배경색이 된다
- Source의 alpha값을 0.0에서 1.0까지 증가시켜 보자

Destination (원래 색)	glClearColor(0.0, 0.0, 1.0, 0.0);	파란색
Source	glColor4f(1.0, 0.0, 0.0, 0.0);	빨간색
Source의 alpha값	0.0	완벽히 투명

- 예: 교재 코드 13장
- run “blendRectangles1.cpp”, which draws two translucent rectangles with their alpha value equal to 0.5 set by calss of the form glColor4f(*, *, *, 0.5), the red one being closer to the viewer than the blue one.
- The code order in which the rectangles are drawn can be toggled by pressing space

combined instead.

As $src_A = 1 - src_A = 0.5$, no matter which rectangle is drawn first, the blending equation (13.3) gives simply

$$(dst_R, dst_G, dst_B, dst_A) = 0.5 * (src_R, src_G, src_B, src_A) + 0.5 * (dst_R, dst_G, dst_B, dst_A) \quad (13.4)$$

which is symmetric in source and destination. So, one may wonder why the one drawn second seems to dominate where the images intersect. The reason is that the rectangle drawn first is blended with the background white, diluting its color, while the second-drawn rectangle comes in at "full strength". Let's check that this is actually so in the next example.

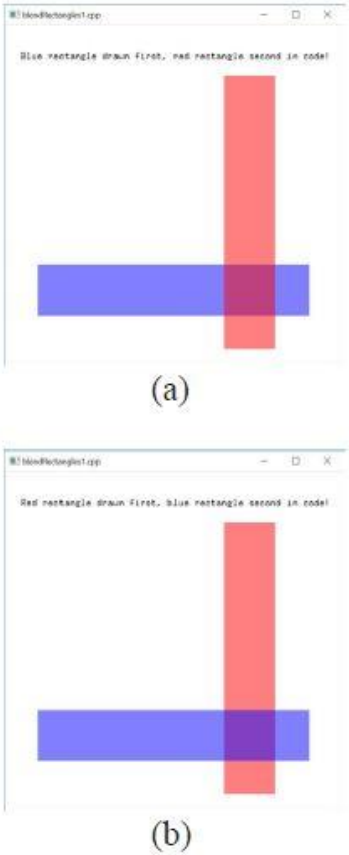


Figure 13.2: Screenshot of `blendRectangles1.cpp` with (a) the blue rectangle first in code (b) the red rectangle first in code.