

# DCC192

2025/1



# Desenvolvimento de Jogos Digitais

A26: Gráficos 3D — Introdução

Prof. Lucas N. Ferreira

# Plano de Aula



- ▶ Computação Gráfica
- ▶ Modelos 3D
  - ▶ Vértices e Atributos
  - ▶ Formatos de especificação
- ▶ Pipeline Gráfico
  - ▶ Shaders de Vértice e Fragmentos
- ▶ Visão geral de um programa OpenGL/GLSL
  - ▶ Compilando Shaders
  - ▶ Contexto OpenGL e Buffers

# Computação Gráfica



Um dos principais problemas da área de Computação Gráfica consiste em gerar uma *imagem* (i.e., arranjo bidimensional de pixels) a partir de uma cena composta por:

▶ **Objetos 3D**

Geralmente representados por conjuntos de vértices.

▶ **Câmera**

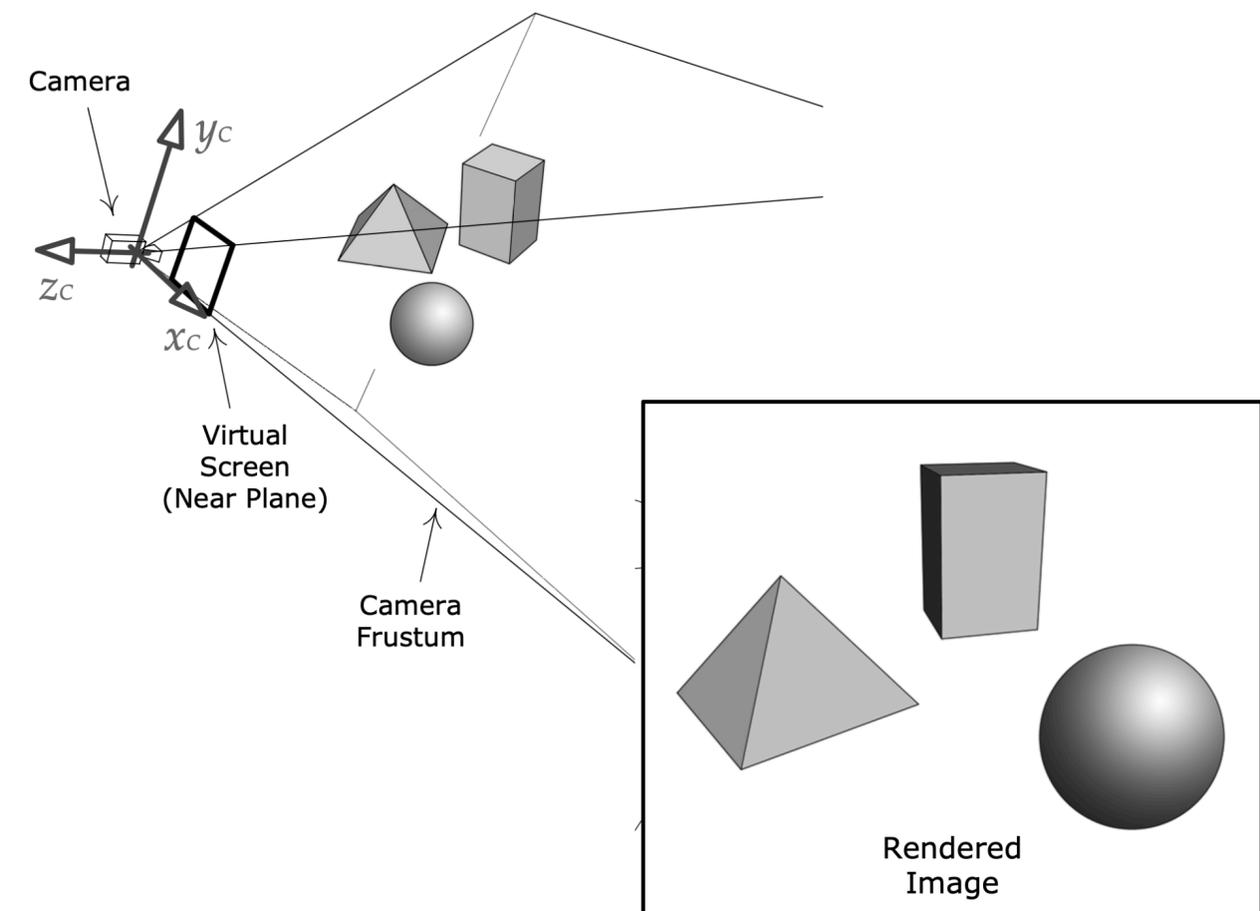
Geralmente representada por uma posição, orientação, distância focal e planos de recorte próximo e distante

▶ **Fonte de Luz**

Vários tipos de fontes de luz podem ser especificadas: direcional, ambiente e spot.

▶ **Materiais**

Propriedades visuais dos objetos, descrevendo como a luz deve interagir com os objetos.



# Renderização em Tempo Real vs. Pré-Renderização



Há uma importante distinção dentro da Computação Gráfica quanto às restrições de tempo impostas no processo de geração de imagens:



Marvel's Spider-Man (2018)

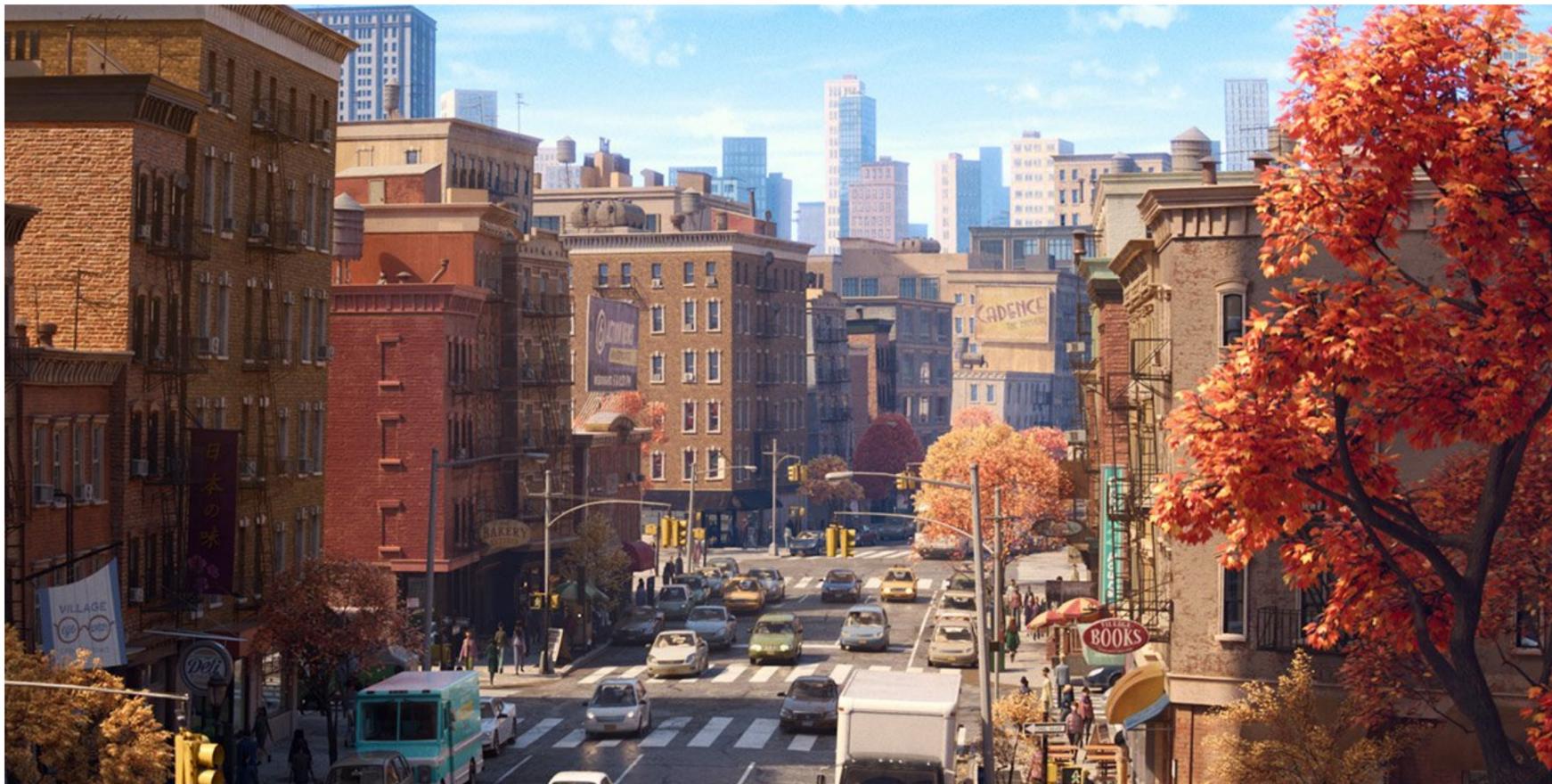
## Renderização em Tempo Real

- ▶ As imagens são geradas instantaneamente, muitas vezes a 60 quadros por segundo.
- ▶ Algoritmos e técnicas priorizam a velocidade, mesmo que isso signifique sacrificar um pouco da qualidade visual.
- ▶ Possibilita interatividade no processo de geração de imagens.

# Renderização em Tempo Real vs. Pré-Renderização



Há uma importante distinção dentro da Computação Gráfica quanto às restrições de tempo impostas no processo de geração de imagens:



Pixar's Soul – Film

## Pré-Renderização

- ▶ As imagens são geradas com antecedência, antes de serem mostradas ao usuário.
- ▶ Algoritmos e técnicas priorizam qualidade visual máxima, com efeitos realistas de luz, sombra, reflexo, refração, etc.
- ▶ É usada quando o conteúdo não precisa ser interativo, como em filmes.

# Objetos 3D

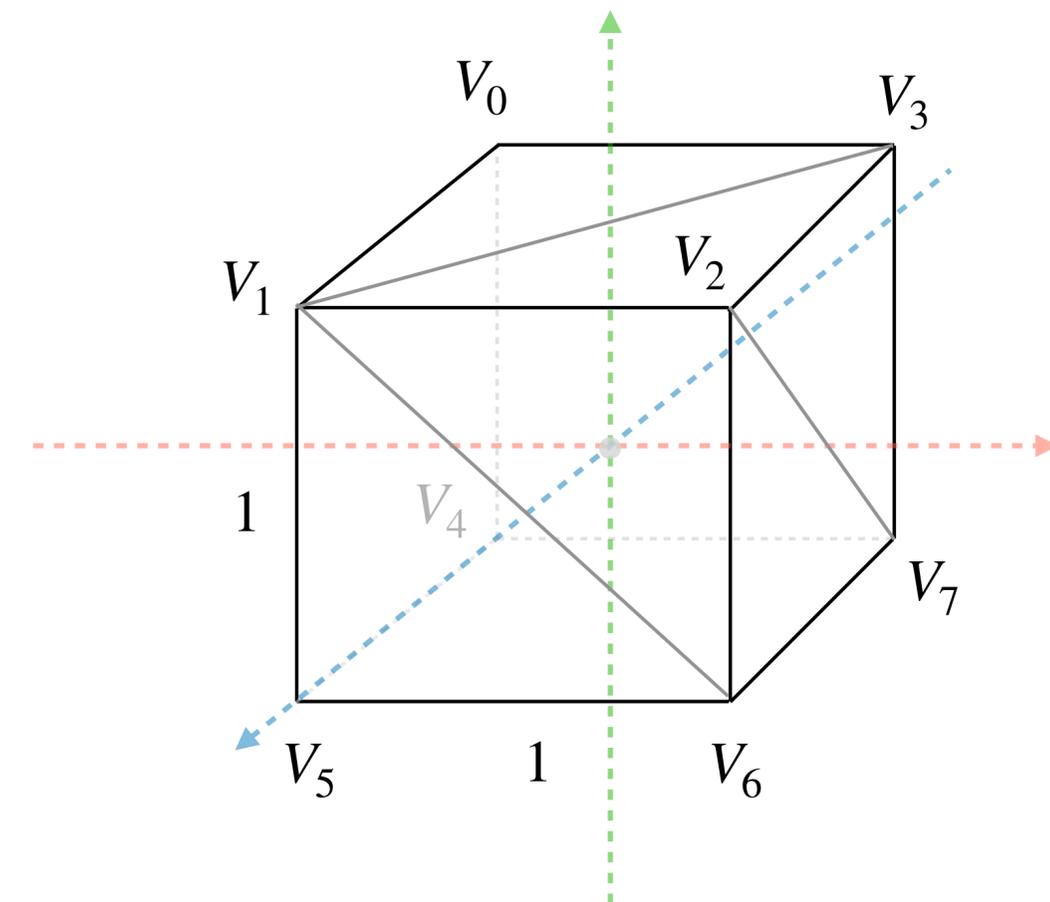


A maioria dos objetos (ou modelos) em jogos são representados por malhas de triângulos. A forma mais simples de definir uma malha é utilizar uma lista de triângulos:

- ▶ Cada triângulo é formado por um grupo de três vértices adjacentes  $\{V_1, V_2, V_3\}$ ,  $V_i \in \mathbb{R}^3$

$V_0$	$V_1$	$V_3$	$V_1$	$V_2$	$V_3$	$V_0$	$V_5$	$V_1$	...	$V_5$	$V_7$	$V_6$
0	1	2	3	4	5	6	7	8	...	10	11	35

- ▶ Os vértices são definidos no **espaço do objeto**, um sistema de coordenadas centrado no objeto. Por exemplo, no cubo ao lado:
  - ▶  $V_1 = [-0.5, 0.5, 0.5]$
  - ▶  $V_3 = [0.5, 0.5, -0.5]$
  - ▶  $V_5 = [-0.5, -0.5, 0.5]$



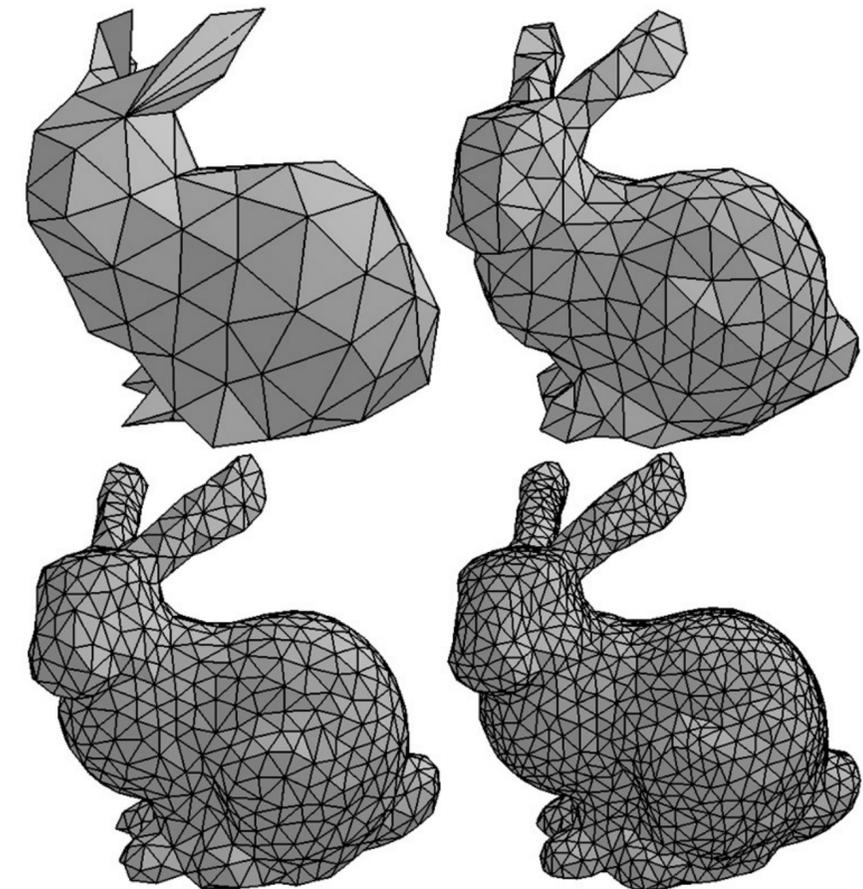
Por exemplo, um cubo pode ser representado por 12 triângulos, dois para cada face.

# Por que triângulos?



Em jogos digitais, malhas triângulos são utilizados para representar os objetos 3D pois triângulos são:

- ▶ O tipo de polígono mais simples;
- ▶ Sempre planares;
- ▶ Permanecem triângulos sob a maioria dos tipos de transformações;
- ▶ A maioria das GPUs para jogos são projetadas em torno da rasterização triangular.

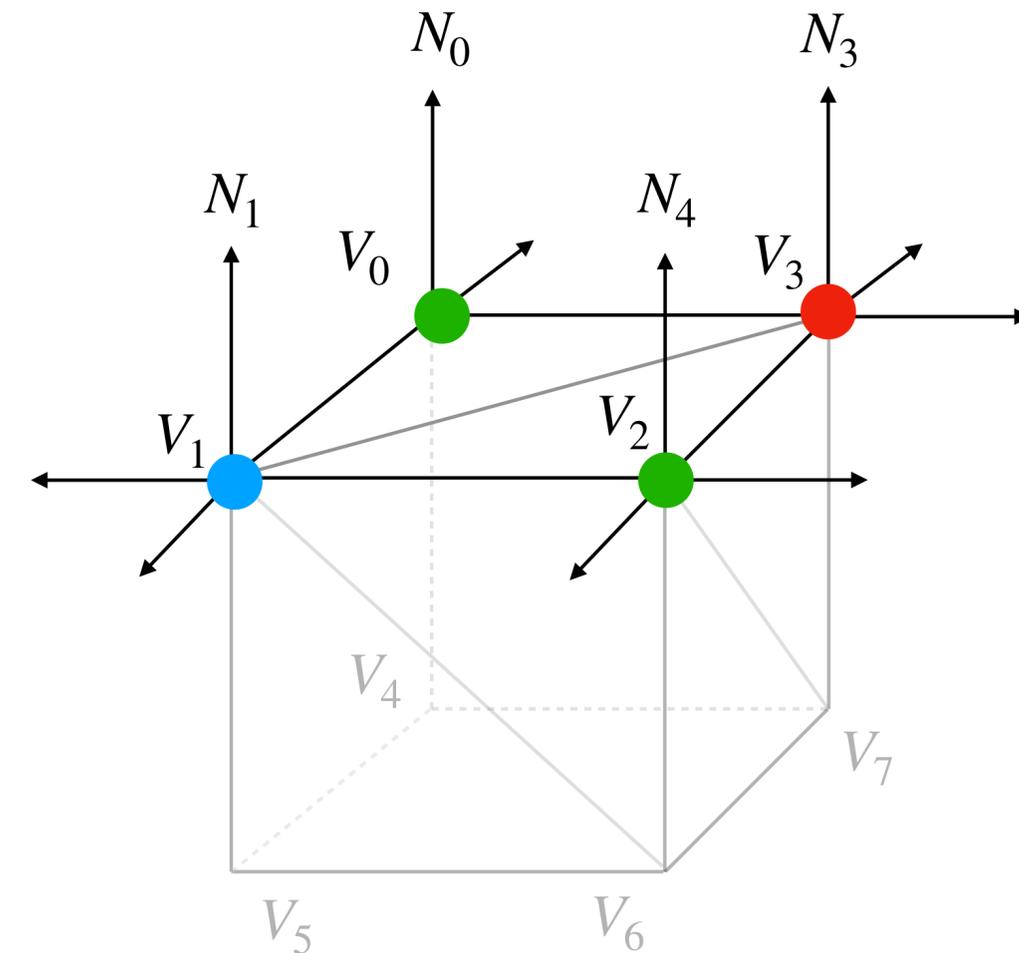


# Atributos dos Vértices



Além de suas posições, cada vértice  $V_i$  possuem atributos associados, para definir propriedades visuais de cada região do objeto:

- ▶ Posição:  $V_i = [v_{ix}, v_{iy}, v_{iz}]$
- ▶ Vetor normal:  $N_i = [n_{ix}, n_{iy}, n_{iz}]$
- ▶ Cor Difusa:  $D_i = [d_{iR}, d_{iG}, d_{iB}, d_{iA}]$
- ▶ Cor Especular:  $S_i = [s_{iR}, s_{iG}, s_{iB}, s_{iA}]$
- ▶ Coordenadas de Texturas:  $U_i = [u_{ij}, v_{ij}]$
- ▶ ...

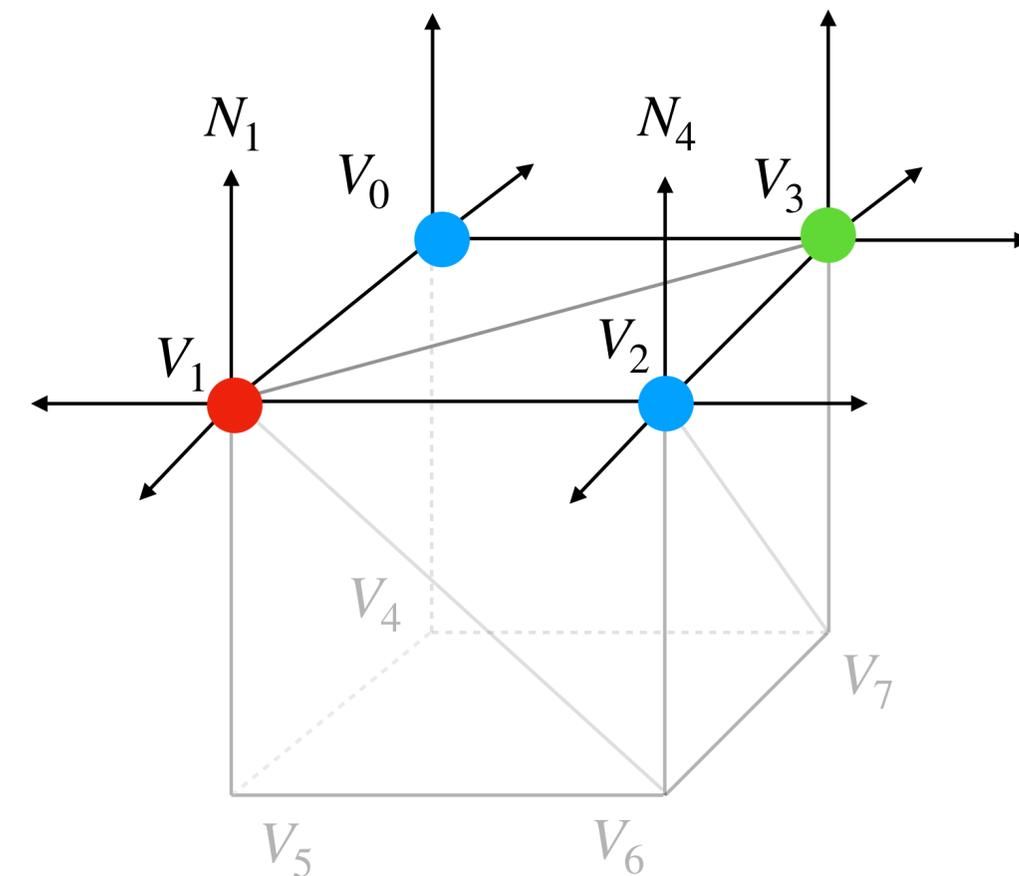
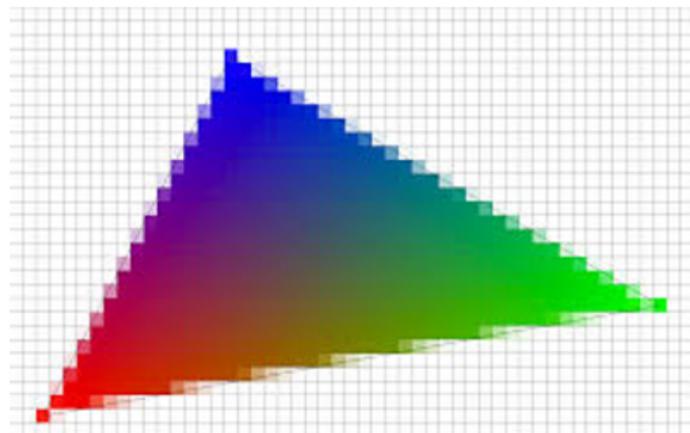


# Atributos dos Vértices – Interpolação



Como especificamos apenas os atributos dos vértices, mas precisamos colorir as superfícies inteiras, a GPU interpola linearmente os valores dos atributos entre as superfícies:

- ▶ Todos os atributos são interpolados: posição, vetor normal, cor, etc...
- ▶ Exemplo de como a interpolação de cor afeta a superfície do triângulo  $\{V_0, V_1, V_3\}$ :



# Criando Modelos



Modelos 3D para jogos são tipicamente criados com editores gráficos especializados, por exemplo:

- ▶ Blender  
<https://www.blender.org/>
- ▶ Autodesk Maya  
<https://www.autodesk.com/maya>
- ▶ ZBrush  
<https://www.maxon.net/zbrush>
- ▶ BlockBench  
<https://www.blockbench.net/>



[https://www.youtube.com/watch?v=Y05txBWzBqE&ab\\_channel=Lukky](https://www.youtube.com/watch?v=Y05txBWzBqE&ab_channel=Lukky)

# Salvando/Carregando Modelos



Os editores de modelos 3D exportam objetos para arquivos do tipo FBX, OBJ, STL, etc. Esses formatos armazenam os vértices, as faces, as coordenadas de texturas, entre outros:

```
# OBJ file format with ext .obj
# vertex count = 2503
# face count = 4968
v -3.4101800e-003 1.3031957e-001 2.1754370e-002
v -8.1719160e-002 1.5250145e-001 2.9656090e-002
v -3.0543480e-002 1.2477885e-001 1.0983400e-003
v -2.4901590e-002 1.1211138e-001 3.7560240e-002
v -1.8405680e-002 1.7843055e-001 -2.4219580e-002
v 1.9067940e-002 1.2144925e-001 3.1968440e-002
...
f 1069 1647 1578
f 1058 909 939
f 421 1176 238
f 1055 1101 1042
f 238 1059 1126
f 1254 30 1261
```



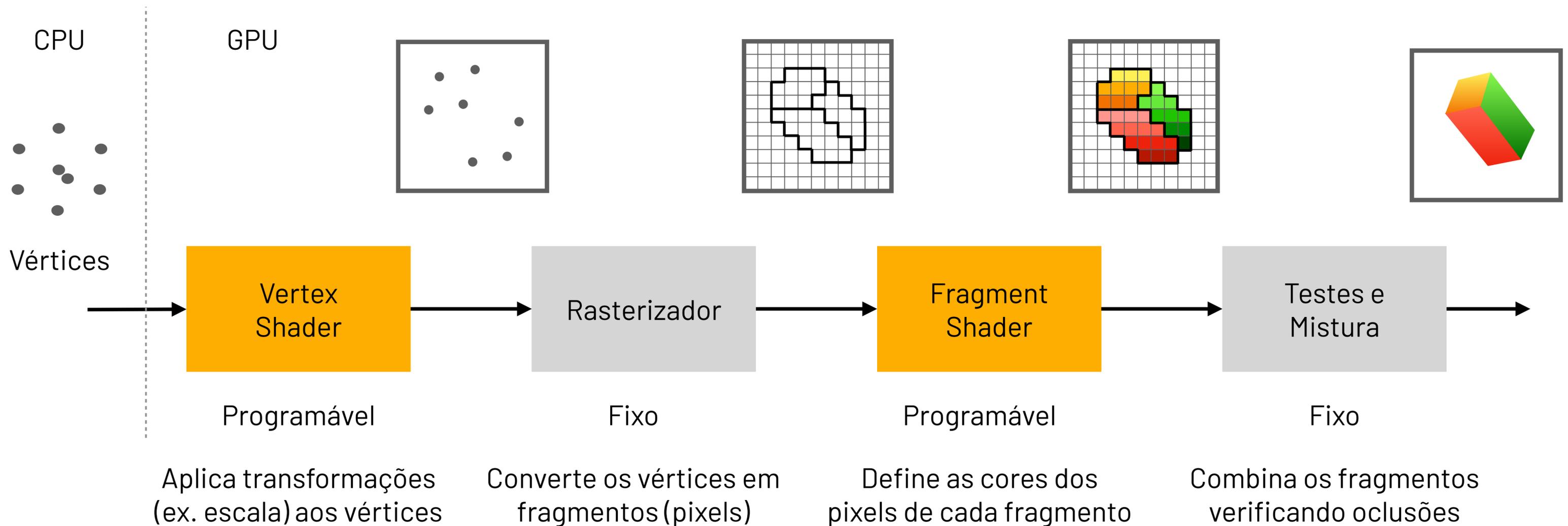
<https://graphics.stanford.edu/~mdfisher/Data/Meshes/bunny.obj>

[https://en.wikipedia.org/wiki/Stanford\\_bunny#/media/File:Stanford\\_Bunny.stl](https://en.wikipedia.org/wiki/Stanford_bunny#/media/File:Stanford_Bunny.stl)

# Pipeline Gráfico



Os frameworks de computação gráfica, como DirectX e OpenGL, geralmente produzem uma imagem a partir de uma cena em uma sequência de operações, chamado de **Pipeline Gráfico**:

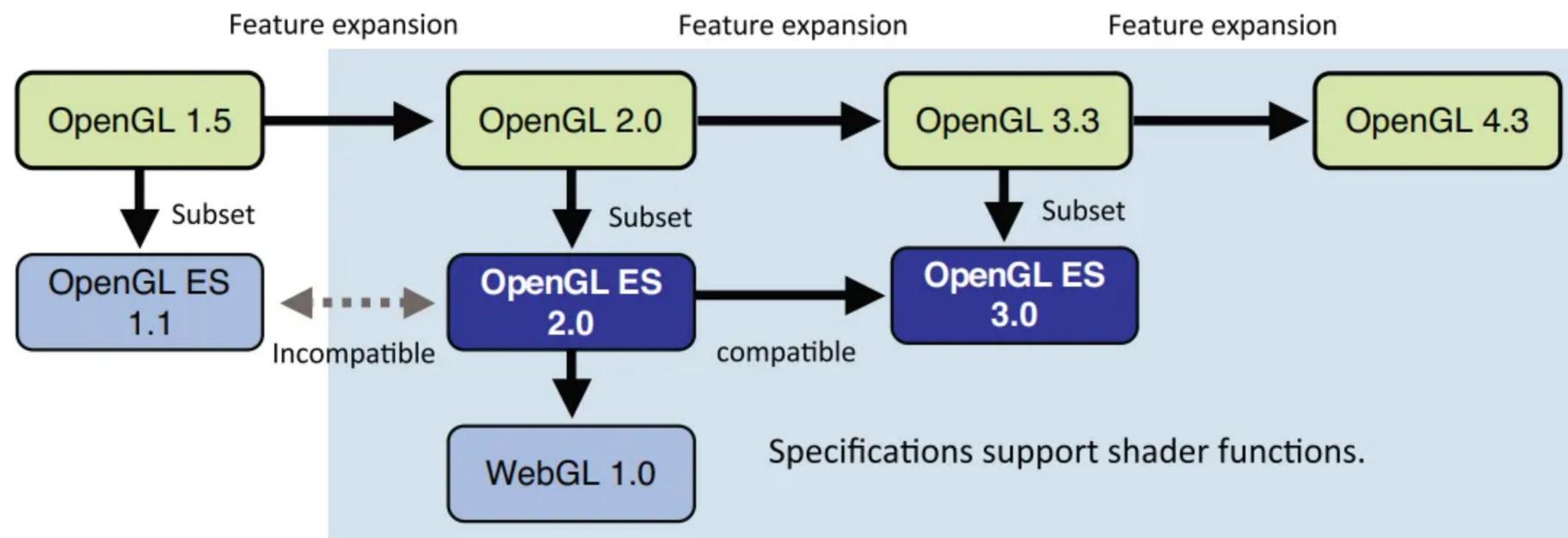


# OpenGL



OpenGL (Open Graphics Library) é uma API multiplataforma para renderização de gráficos 2D e 3D. Ela é uma implementação do pipeline gráfico em GPU:

- ▶ Atua como uma ponte entre o programa e a GPU;
- ▶ Especifica comandos que são enviados à placa gráfica para desenhar objetos na tela.
- ▶ Possui versões para Web (WebGL) e para sistemas embarcados (OpenGL ES)



*Desde a versão 2.0 (2004), permite programar os vertex e fragment shaders com uma linguagem chamada GLSL*

# GLSL



GLSL (OpenGL Shading Language) é a linguagem de programação utilizada para escrever shaders, que são pequenos programas executados diretamente na GPU.

- ▶ Sintaxe similar à linguagem de programação C
- ▶ Cada shader é compilado e ligado via comandos OpenGL (`glCompileShader`, `glAttachShader`, `glLinkProgram`, etc).

```
constexpr std::string_view vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec2 aPos;
void main() {
    gl_Position = vec4(aPos, 0.0, 1.0);
}
)";
```

Exemplo de Vertex Shader em GLSL

```
constexpr std::string_view fragmentShaderSource = R"(
#version 330 core
out vec4 FragColor;
void main() {
    FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
)";
```

Exemplo de Fragment Shader em GLSL

# Vertex Shader



O Vertex Shader é um pequeno programa que roda na GPU para aplicar transformações aos vértices de entrada.

- ▶ É escrito como uma string dentro do programa principal (ex. main.cpp)
- ▶ É compilado/linkado pela CPU e enviado pela GPU usando funções OpenGL (em tempo de execução)

```
constexpr std::string_view vertexShaderSource = R"(  
#version 330 core  
  
layout (location = 0) in vec2 aPos;  
  
void main() {  
    gl_Position = vec4(aPos, 0.0, 1.0);  
}  
)";
```

Define a versão do GLSL usada

Declara uma variável de entrada do tipo vec2

Ponto de entrada do shader.

`gl_Position` é uma variável predefinida da OpenGL, que armazena a posição final do vértice.

*Alterar a variável `gl_Position` é equivalente a retornar o valor do vértice transformado (GLSL não usa return)*

# Fragment Shader



O Fragment Shader é um pequeno programa que roda na GPU para definir a cor final de cada pixel do fragmentz.

- ▶ É escrito como uma string dentro do programa principal (ex. main.cpp)
- ▶ É compilado/linkado pela CPU e enviado pela GPU usando funções OpenGL (em tempo de execução)

```
constexpr std::string_view fragmentShaderSource = R"(
#version 330 core
out vec4 FragColor;
void main() {
    FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
)";
```

Define a versão do GLSL usada

Declara uma variável de saída do tipo vec4

Ponto de entrada do shader

FragColor representa a cor final que será enviada ao framebuffer (a tela, no caso).

*Alterar a variável `FragColor` é equivalente a retornar a cor do vértice (GLSL não usa return)*

# OpenGL em SDL



Para usar OpenGL com SDL2, precisamos criar um contexto OpenGL dentro de uma janela SDL:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);

SDL_Window* window = SDL_CreateWindow("OpenGL Triangle", 100, 100, 800, 600, SDL_WINDOW_OPENGL);
if (!window) {
    std::cerr << "Failed to create window: " << SDL_GetError() << std::endl;
    return -1;
}

SDL_GLContext context = SDL_GL_CreateContext(window);
if (!context) {
    std::cerr << "Failed to create OpenGL context: " << SDL_GetError() << std::endl;
    return -1;
}

glewExperimental = GL_TRUE;
if (glewInit() != GLEW_OK) {
    std::cerr << "Failed to initialize GLEW\n";
    return -1;
}
```

# O que é a GLEW?



GLEW (OpenGL Extension Wrangler Library) é uma biblioteca em C/C++ usada para carregar e acessar funções modernas do OpenGL.

## Por que ela é necessária?

- ▶ O OpenGL moderno (versões 2.0 em diante) adicionou centenas de funções novas, que não estão disponíveis diretamente nos headers padrão (<GL/gl.h>), especialmente em sistemas como Windows.
- ▶ Por exemplo, se você usar função `glGenBuffers()`, pode receber um "undefined reference".
- ▶ Isso acontece porque o endereço real da função está na GPU, e precisa ser carregado.

## O que ela faz?

- ▶ Carrega automaticamente os ponteiros de função da GPU.
- ▶ Permite que você use qualquer função moderna do OpenGL como `glCreateShader`, `glBindVertexArray`, etc.

# Próxima aula



## A27: Gráficos 3D II

- ▶ Mais detalhes de OpenGL/GLSL
- ▶ Especificando Múltiplos Objetos
  - ▶ Sistema de Coordenadas do mundo
  - ▶ Transformações
- ▶ Cores e Texturas