UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

Disciplina: [**GDSCO0051] Introdução à Computação Gráfica - Turma 01**.
Professor: **Christian Azambuja Pagot (email: christian@ci.ufpb.br)**.
Data de entrega: **23/04/2019**.

**Trabalho II**

# Objetivo

O objetivo deste trabalho é familiarizar os alunos com a estrutura e o funcionamento do *pipeline* gráfico através da implementação de um *pipeline* completo, capaz de transformar vértices descritos no espaço do objeto em primitivas rasterizadas no espaço de tela.

# Atividade

O sistema de desenvolvido no trabalho I (TI1) implementa um dos últimos estágios do pipeline gráfico, responsável pela rasterização das primitivas no espaço de tela. Neste sistema, a descrição dos modelos é feita através de coordenadas inteiras definidas sobre o espaço bidimensional discretizado da tela.

Esta forma de descrever primitivas, entretanto, não é prática pois, em muitos casos, os modelos geométricos são tridimensionais e se extendem para além dos limites definidos pela tela. De forma a facilitar o processo de descrição de modelos, opta-se por fazê-lo em um espaço mais adequado, tanto em termos de extensão quanto de dimensões, chamado de *espaço do objeto*. Após a descrição no espaço do objeto, o modelo é submetido a uma série de transformações que resultam em seu mapeamento final para o espaço da tela. A imagem final então é obtida através da rasterização das primitivas projetadas na tela.

Esta segunda atividade consiste na implementação das transformações que levarão os vértices descritos originalmente no espaço do objeto para o espaço de tela. A rasterização das primitivas será feita pelo código previamente desenvolvido no trabalho TI1.

# Desenvolvimento

Os alunos deverão implementar todas as transformações do *pipeline* em forma de matriz e utilizando coordenadas homogêneas. Abaixo são enumeradas as etapas normalmente encontradas ao longo de um *pipeline* gráfico. Cada etapa é acompanhada de uma breve descrição de suas características, dados de entrada e de saída.

1. **Transformação: Espaço do Objeto → Espaço do Universo (Implementar)**
   Esta etapa do *pipeline* é responsável por transformar vértices, originalmente descritos no espaço do objeto, para o espaço do universo. Isto é feito através da multiplicação destes vértices por uma matriz denominada *matriz de modelagem (*ou *model matrix)*. A matriz de modelagem é composta por uma sequência de transformações geométricas que posicionam o modelo no universo. As transformações que podem compor a matriz de modelagem são a rotação, translação, escala e o cisalhamento (*shear*).

2. **Transformação: Espaço do Universo → Espaço da Câmera (Implementar)**
   Esta etapa do *pipeline* é responsável por transformar vértices do espaço do universo para o espaço da câmera. Isto é feito através da multiplicação dos vértices por uma matriz denominada *matriz de visualização (*ou *view matrix)*. A matriz de visualização é composta por uma translação e uma rotação. Esta translação e esta rotação são definidas a partir das informações da câmera (posição, direção e "up").

3. **Transformação: Espaço da Câmera → Espaço Projetivo ou de Recorte (Implementar)**
   Esta etapa do *pipeline* é responsável por transformar vértices do espaço da câmera para o espaço de recorte (ou projetivo). Isto é feito através da multiplicação dos vértices por uma matriz denominada de matriz de projeção (ou *projection matrix*). Após esta transformação, a coordenada homogênea dos vértices pode (e provavelmnente irá) apresentar valores diferentes de 1.

4. **Transformação: Espaço de Recorte → Espaço "Canônico" (Implementar)**
   Esta etapa do *pipeline* é responsável por transformar pontos do espaço de recorte para o espaço canônico (ou NDC – *Normalized Device Coordinates*). Isto é feito em duas etapas. Primeiro, dividem-se as coordenadas dos vértices no espaço de recorte pela sua coordenada homogênea. Esta transformação gera uma alteração da geometria da cena, tornando menores os objetos que estiverem mais distantes da câmera. Adicionalmente, esta transformação mapeia o volume de visualização (*view frustum*), com formato piramidal, em um hexahedro. A seguir, os vértices são multiplicados por uma matriz adicional que, ao aplicar escalas e translações, transforma estes vértices para o espaço canônico, transformando o hexahedro anterior em um cubo de coordenadas unitárias e centrado na origem.

5. **Transformação: Espaço Canônico → Espaço de Tela (Implementar)**
   Esta etapa do *pipeline* é responsável por transformar pontos do espaço canônico para o espaço de tela. Isto é feito através da multiplicação dos vértices por uma matriz chamada ViewPort, que contém escalas e translações.

6. **Rasterização (Já foi implementada no TI1!)**
   Esta etapa gera a rasterização dos modelos no espaço de tela.

Após o término da implementação, os alunos deverão comparar os resultados obtidos com os resultados gerados por uma aplicação equivalente escrita em OpenGL. Os dois sistemas devem gerar os mesmos resultados.

**OBS 1**: A câmera perspectiva do OpenGL apresenta alguns parâmetros extras que a câmera sintética a ser implementada não possui. Estes parâmetros são o *aspecto* e o ângulo de visão (FOV – *Field of View*). Abaixo seguem dicas de como lidar com estas limitações da câmera a ser implementada:

1. **Aspecto:** Aspecto é a razão entre a largura e a altura da imagem final (ou da *viewport*). Como estamos utilizando sempre *viewports* quadradas (mesma largura e altura), o aspecto é sempre 1. Assim, quando for criada a câmera em OpenGL para comparação, deve-se setar o aspecto para 1.

2. **FOV:** A câmera sintética a ser implementada não possui o parâmetro FOV. Entretanto, é possível se obter um efeito equivalente ao do FOV da câmera OpenGL ajustando-se o

parâmetro *d* (a distância do ViewPlane) da matriz de projeção.

Este trabalho já vem acompanhado de uma biblioteca simples para a carga de modelos descritos em formato OBJ (*Wavefront .obj*). Entretanto, os alunos são encorajados a utilizarem outras bibliotecas mais sofisticadas para carga de modelos caso seja necessário (*e.g.* Assimp).

Os alunos devem fazer comparações entre os resultados gerados pelos seus renderizadores e os resultados gerados pelo OpenGL através da renderização de modelos geométricos mais complexos. O próprio arquivo da biblioteca já vem acompanhado de um modelo (*monkey_head2.obj*) que pode ser utilizado nos testes de comparação. Incentiva-se que o aluno também faça testes com outros modelos.

O Apêndice I deste trabalho contém uma breve descrição de como exportar arquivos OBJ a partir do Blender.

O Apêndice II deste trabalho contém um trecho de código escrito em C++ que carrega modelos geométricos a partir de arquivos externos através da biblioteca Assimp (http://www.assimp.org) (caso os alunos desejem utilizá-la).

**OBS 1:** Todos os sistemas de coordenadas, com exceção do espaço de tela, seguem a convenção RHS (mão direita).

**OBS 2:** Não é permitido o uso de nenhuma biblioteca externa, com a exceção de:

- glut, SDL, GLFW, e assemelhados.
- OpenGL
- *loader* de arquivos OBJ fornecido junto com o trabalho, ou algum outro *loader* (como o Assimp - http://www.assimp.org)
- uma biblioteca para operação de matrizes e vetores (como a Glm - http://glm.g-truc.net).

# Entrega

Prazo: Os trabalhos devem ser entregues até o dia **23/04/2018**, impreterivelmente até as **23 horas e 55 minutos**.O trabalho será considerado entregue assim que estiver acessível no repositório do Github criado para a disciplina.

No *post* deverão constar os *printscreens* dos resultados (acompanhados de todos os parâmetros utilizados na sua geração) e pequenos textos explicativos. Os alunos podem, se desejarem, incluir também um vídeo que ilustre o modelo sendo transformado (rotacionado, escalado, etc.). Abaixo segue um detalhamento de cada um dos componentes que devem aparecer no *post*.

- ***Printscreens***: Devem demonstrar a evolução do processo de implementação, eventuais problemas encontrados, e resultados gerados após suas correções.

- **Texto**:
  - Deve iniciar com uma breve introdução da atividade a ser desenvolvida.
  - Deve explicar, de forma suscinta, as estratégias adotadas pelos alunos na resolução da atividade (estruturas de dados utilizadas e funções desenvolvidas).
  - Apresentar breve discussão sobre os resultados gerados e possvíeis melhoras.
  - Citar eventuais dificuldades encontradas.
  - Listar as referência bibliográficas (livros, artigos, endereços web).

# Avaliação

A avaliação dos trabalhos levará em consideração os seguintes critérios:

- Aderência ao tema proposto.
- Clareza, organização, corretude, legibilidade e capacidade de síntese do texto.
- Relevância dos elementos apresentados (imagens, trechos de código, etc.) no contexto das discussões.
- Embasamento técnico dos argumentos.
- Eficiência e eficácia das soluções apresentadas.
- Capacidade de análise dos resultados e autocrítica.

**Importante:**

- Não serão aceitos trabalhos atrasados ou que apresentem indícios de plágio.
- Não serão aceitos trabalhos enviados por qualquer outro meio que não o repositótio informado previamente pelos alunos.

# Apêndice I

O software de modelagem 3D *Blender* (versão 2.79 ou menor) é capaz de salvar seus modelos em diversos formatos, incluindo o *Wavefront .obj*. Uma vez que o modelo esteja finalizado, para que se possa exportá-lo, deve- se seguir os seguintes passos:

1. Selecionar somente o objeto que se deseja exportar.
2. Selecionar no menu principal *File/Export/Wavefront (.obj)* (→ **Figura 1**).
3. No painel de opções que surgirá à esquerda (→ **Figura 2**) **marcar apenas** os ítens: *Selection Only*, *Triangulate Faces*, *Objects as OBJ Objects*.
4. Para que o objeto não seja exportado em um sistema de coordenadas diferente do utilizado na aplicação (RHS), certifique-se de que abaixo deste painel a direção seja *-Z Forward*, e que o Up seja *Y Up*.

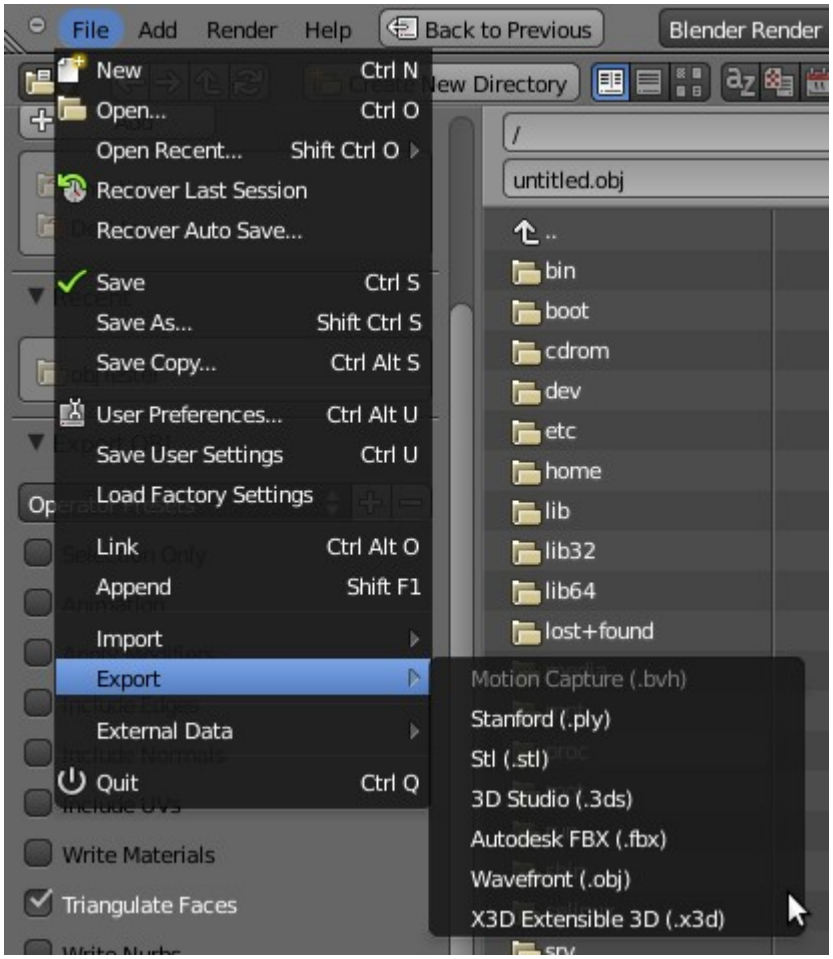

*Figura 1 - Menu principal.*

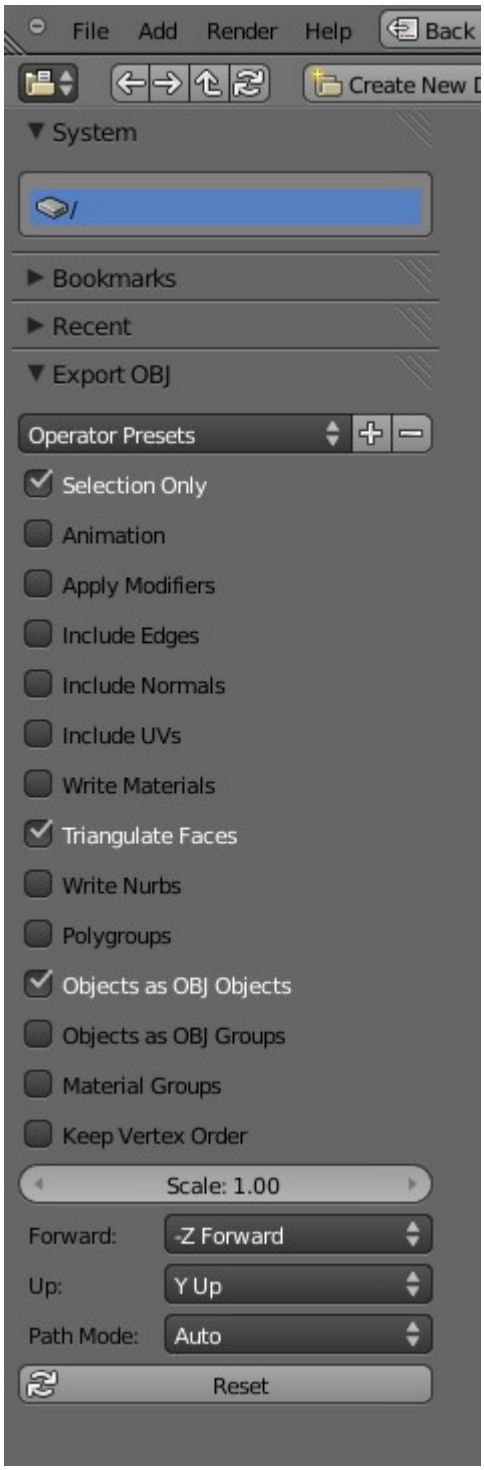


*Figura 2 - Painel de opções para exportação de arquivo OBJ.*

# Apêndice II

Carga de um modelo geométrico, a partir de um arquivo externo, através da biblioteca Assimp (código de exemplo em C++):

```
// file_name contém o nome do arquivo a ser carregado
int loadMesh( const std::string &file_name )
{
  std::ifstream fin( file_name.c_str() );

  if( !fin.fail() )
    fin.close();
  else
  {
    std::cerr << "Couldn't open file: " << file_name << std::endl;
    return EXIT_FAILURE;
  }

  Assimp::Importer assimp_importer;

  assimp_scene_ = assimp_importer.ReadFile( file_name,
                                            aiProcess_Triangulate );

  if( !assimp_scene_ )
  {
    std::cerr << assimp_importer.GetErrorString() << std::endl;
    return EXIT_FAILURE;
  }

  if( assimp_scene_->HasMeshes() )
  {
    for( unsigned int mesh_id = 0; mesh_id < assimp_scene_->mNumMeshes; mesh_id++ )
    {
      const aiMesh *mesh_ptr = assimp_scene_->mMeshes[mesh_id];

      for( unsigned int vertex_id = 0; vertex_id < mesh_ptr->mNumVertices;
vertex_id += 3 )
      {
        const aiVector3D *vertex_ptr = &mesh_ptr->mVertices[vertex_id];

        glm::dvec3 v0{ vertex_ptr[0].x, vertex_ptr[0].y, vertex_ptr[0].z };
        glm::dvec3 v1{ vertex_ptr[1].x, vertex_ptr[1].y, vertex_ptr[1].z };
        glm::dvec3 v2{ vertex_ptr[2].x, vertex_ptr[2].y, vertex_ptr[2].z };

        // ---> Aqui você salva os vértices V0, V1 e V2 do
        // ---> triângulo na sua estrutura de dados!!!

      }
    }
  }

  return EXIT_SUCCESS;
}
```