Universidade Federal de Mato Grosso do Sul
Faculdade de Computação

Computação Gráfica 2019

Trabalho – Parte 2

Paulo A. Pagliosa

O propósito da **Parte 1** foi implementar a hierarquia de objetos de cena de uma cena, em que um objeto de cena pode conter uma coleção de objetos de cenas filhos e uma coleção de componentes que caracterizam suas funcionalidades. Dois tipos de componentes foram usados: `Transform` e `Primitive`. O primeiro representa uma transformação resultante da combinação de uma escala, de uma rotação e de uma translação (TRS) aplicadas ao objeto de cena. O segundo representa um ator da cena com geometria modelada por uma malha de triângulos. Os objetivos da **Parte 2** incluem adicionar funcionalidades à interface gráfica do programa, finalizar a implementação da classe `Transform` e introduzir um novo componente de objetos de cena: `Camera`. Uma câmera, conforme visto no Capítulo 1 e em detalhes no Capítulo 5, define um volume de vista da cena visível a dado expectador.

Vamos começar por `Transform`. Como se pode notar na janela `Inspector` da GUI, os atributos editáveis do componente são posição, rotação e escala. A posição define a origem de um sistema de coordenadas locais do objeto de cena, A, que contém o componente, em relação ao sistema de coordenadas locais do objeto de cena $\mathrm{pai}(A)$, o pai de A. A rotação define a orientação dos eixos do sistema local de A em relação aos eixos do sistema local de seu pai, em termos de *ângulos de Euler* (a_x, a_y, a_z), os quais expressam rotações de a_z, a_x e a_y graus em torno dos eixos z, x e y, respectivamente, do sistema local de A, *nesta ordem*[1]. Portanto, posição, rotação e escala compõem uma matriz TRS, $\mathbf{L}_A$, que transforma pontos em coordenadas locais de A para coordenadas locais de $\mathrm{pai}(A)$. A matriz que transforma pontos em coordenadas locais de A para coordenadas globais é

$$\mathbf{G}_A = \mathbf{G}_{\mathrm{pai}(A)} \cdot \mathbf{L}_A, \tag{1}$$

em que $\mathbf{G}_{\mathrm{pai}(A)}$ é a matriz que transforma pontos em coordenadas locais de $\mathrm{pai}(A)$ para coordenadas globais. (Se A for um objeto de cena da raiz da cena, então $\mathbf{G}_A = \mathbf{L}_A$). De forma inversa, a matriz que transforma pontos em coordenadas globais para coordenadas locais de A é

$$\mathbf{G}_A^{-1} = \left(\mathbf{G}_{\mathrm{pai}(A)} \cdot \mathbf{L}_A\right)^{-1} = \mathbf{L}_A^{-1} \cdot \mathbf{G}_{\mathrm{pai}(A)}^{-1}, \tag{2}$$

em que $\mathbf{L}_A^{-1}$ é a matriz que transforma pontos em coordenadas locais de $\mathrm{pai}(A)$ para coordenadas locais de A. Da Equação (1), tem-se ainda que

$$\mathbf{L}_A = \mathbf{G}_{\mathrm{pai}(A)}^{-1} \cdot \mathbf{G}_A. \tag{3}$$

Se A contiver um componente do tipo `Primitive`, então $\mathbf{G}_A$ é a matriz de modelo, $\mathbf{M}_{\mathrm{m}}$, do primitivo, isto é, que transforma as coordenadas locais, $(x_o, y_o, z_o) \in \mathbb{E}^3$, de um vértice da malha de triângulos do primitivo em coordenadas globais. Para um componente

[1] Ângulos de Euler podem ser convertidos em um quatérnio (e vice-versa), o qual pode ser convertido em uma matriz de rotação (e vice-versa). Veja a implementação em `Quaternion.h` e `Matrix3x3.h` no diretório `cg/common/include/math`.

do tipo `Camera` contido no objeto de cena A, a escala não tem efeito algum, enquanto que a posição e a rotação embutidas em $\mathbf{G}_A^{-1}$ definem a matriz de vista, $\mathbf{M}_{\mathrm{v}}$, da câmera. Esta transforma as coordenadas globais de um ponto em coordenadas de vista, ou de câmera, relativas ao sistema de coordenadas locais da câmera (VRC). Em adição, a câmera fornece a matriz de projeção, $\mathbf{M}_{\mathrm{p}}$, responsável por transformar coordenadas de câmera de um ponto em coordenadas de recorte $[x_r, y_r, z_r, w_r] \in \mathbb{T}_3$. Combinando as três transformações, temos

$$\begin{bmatrix} x_r \\ y_r \\ z_r \\ w_r \end{bmatrix} = \mathbf{M}_{\mathrm{mvp}} \cdot \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}, \tag{4}$$

em que

$$\mathbf{M}_{\mathrm{mvp}} = \mathbf{M}_{\mathrm{vp}} \cdot \mathbf{M}_{\mathrm{m}} = \mathbf{M}_{\mathrm{p}} \cdot \mathbf{M}_{\mathrm{v}} \cdot \mathbf{M}_{\mathrm{m}}.$$

As matrizes da Equação (4) são passadas ao programa GLSL fornecido nesta parte do trabalho. Estas são usados no *shader* de vértice para determinação das coordenadas de recorte requeridas pela OpenGL. Como resultado, todo vértice da cena que estiver no interior do volume de vista será mapeado para o interior do NDC. O código fornecido computa as matrizes $\mathbf{M}_{\mathrm{m}}$ e $\mathbf{M}_{\mathrm{vp}}$, além de implementar métodos de manipulação da câmera da vista da cena que permitam ao usuário modificar interativamente (com uso do teclado e/ou mouse) a posição, orientação, dimensões e forma do volume de vista. Tais métodos incluem a translação, rotação (panorâmica ou orbital), alteração do tamanho da janela de projeção (por exemplo, via *zoom*), e alteração das distâncias dos planos de recorte e do tipo de projeção da câmera.

1 Passo 1: Código-fonte e geração do executável

O código-fonte específico para este exercício está no diretório `cg/p2` do repositório `https://git.facom.ufms.br/pagliosa/cg.git`. O código comum a todas as partes está no diretório `cg/common` do repositório.

Observação: conforme alertado em sala, o código comum a todas as partes pode ser alterado entre um exercício e outro. Assim, você deverá obter a versão mais recente do código diretamente do `git`.

Após atualizar o código comum a todas as partes e baixar o código da **Parte 2** em seu diretório de trabalho, `cg` conterá os diretórios `common`, `p0` (com o seu código da **Parte 0**), `p1` (com o seu código da **Parte 1**) e `p2`. O conteúdo de `p2` será:

```
assets
  meshes
    bunny.obj
    f-16.obj
  shaders
    p2.fs
    p2.vs
build
  vs2019
    imgui.ini
    p2.sln
```

```
    p2.vcxproj
    p2.vcxproj.filters
Assets.cpp
Assets.h
Camera.cpp
Camera.h
Component.h
GLRenderer.cpp
GLRenderer.h
imgui_demo.cpp
Main.cpp
P2.cpp
P2.h
Primitive.h
Renderer.cpp
Renderer.h
Scene.h
SceneEditor.cpp
SceneEditor.h
SceneNode.h
SceneObject.cpp
SceneObject.h
Transform.cpp
Transform.h
```

O diretório `assets` contém o código GLSL dos *shaders* de vértice e fragmento, agora organizados em `assets/shaders`. O diretório `assets/meshes` contém exemplos de arquivos de malhas de triângulos no formato OBJ Wavefront, conforme explicado em sala. Todo arquivo adicionado a este diretório (na versão corrente espera-se que os arquivos em `assets/meshes` sejam OBJ válidos) terá a malha de triângulos correspondente disponível para uso na aplicação. Como no exercício anterior, `build/vs2019` contém o projeto do Visual Studio 2019. Os arquivos listados em azul foram copiados de `p1` e aqueles alterados foram comentadas em sala. Desses, o único que você deve completar é `Transform.cpp`. Os demais deverão ser substituídos com sua implementação da **Parte 1**. A definição e implementação da classe de janela gráfica da aplicação são codificadas, respectivamente, em `P2.h` e `P2.cpp`. Os arquivos `SceneEditor.h` e `SceneEditor.cpp` definem e implementam o (esqueleto do) editor de cena. Este contém a câmera da vista da cena, ou câmera do editor, usada para visualização dos objetos de cena e seus componentes durante a edição da cena. Você poderá alterar esses arquivos se julgar conveniente mover para o editor partes das funcionalidades da janela gráfica que não são específicas da aplicação, como métodos usados na GUI, por exemplo. O código responsável pela renderização com OpenGL da cena, tal como vista por uma câmera de um objeto de cena, deverá ser implementado em `GLRenderer.h` e `GLRenderer.cpp`. Neste exercício, você pode usar o mesmo programa GLSL empregado no editor da cena. (Na **Parte 3**, o programa GLSL será distinto, enquanto que na **Parte 4** haverá um outro tipo de renderizador, baseado no traçado de raios.)

No Visual Studio 2019, abra o arquivo `build/vs2019/p2.sln` e construa o arquivo executável (novamente, você pode fazer isso na configuração de *debug* enquanto desenvolve o exercício, mas não se esqueça que, ao final, o programa executável a ser entregue deve ser construído na configuração de *release*). Você verá, entre outros gerados pelo Visual Studio, o arquivo `p2.exe` no diretório `p2`. A execução do programa exibe em sua tela uma janela

gráfica com novos elementos na interface com o usuário, como mostrado na Figura 1: uma barra de menu principal e duas janelas, uma chamada `Editor View Settings` e outra chamada `Assets`, além das já conhecidas `Hierarchy` e `Inspector`, A demonstração das funcionalidades da interface foi feita em sala.

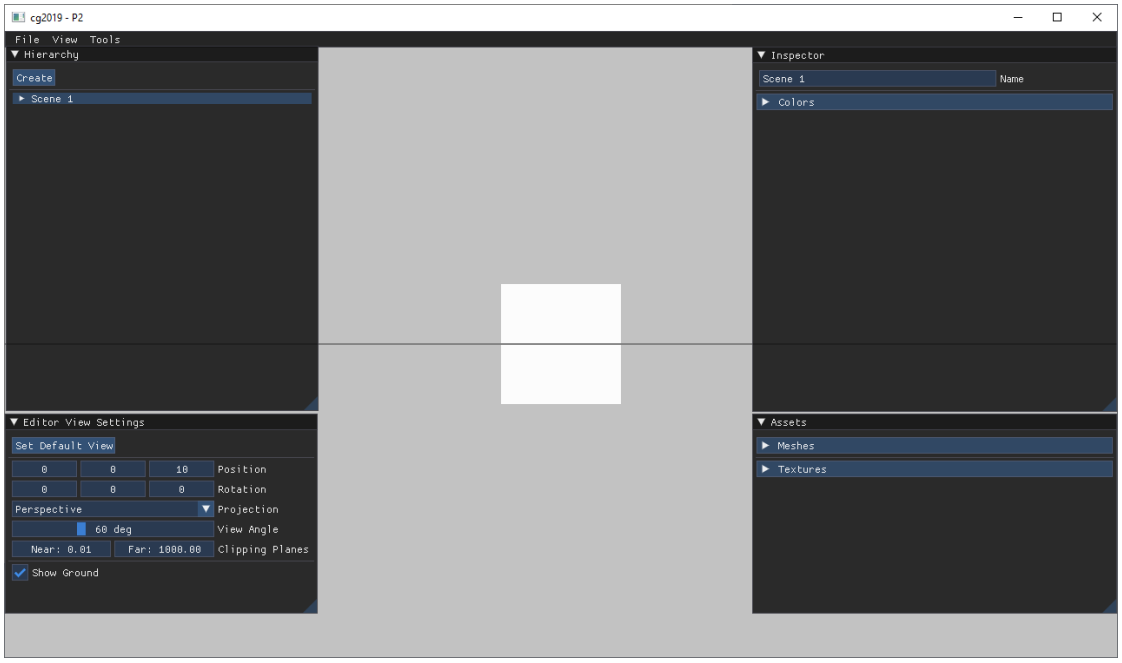

Figura 1: Programa `p2.exe` (no Windows).

2 Passo 2: Tarefas

Antes de iniciar com as atividades específicas da **Parte 2**, você deve primeiro integrar o código que você desenvolveu no exercício anterior com o código fornecido para este exercício. Feito isso, seu programa terá a interface gráfica como ilustrado na Figura 1, mas com sua implementação da hierarquia de objetos de cena e seus componentes funcionando. Assim, você poderá, como esperado desde o exercício anterior, adicionar novos objetos de cena — vazios, contendo um primitivo cuja malha pode ser uma caixa ou esfera e, novidade dessa parte do trabalho, contendo um componente do tipo `Camera` — à hierarquia (e, se você já implementou essa funcionalidade, remover objetos de cena da hierarquia) e inspecionar as propriedades do objeto de cena selecionado. Como demonstrado em sala, com o código fornecido em `p2`, agora você poderá, entre outras funcionalidades:

- visualizar os eixos do sistema local do objeto de cena correntemente selecionado;
- modificar a malha de triângulos associada ao primitivo de um objeto de cena;
- visualizar uma grade no plano xz que serve como referência do "chão" da cena;
- visualizar as malhas de triângulos dos primitivos da cena conforme o ponto de vista definido pela câmera do editor, bem como mover e rotacionar a câmera do editor usando controles da GUI e o teclado e mouse.

Seja A o objeto de cena correntemente selecionado. Até o momento, temos os eixos do sistema local de A renderizados na janela de vista da cena e, se A contiver um primitivo,

então a malha de triângulos do primitivo tem suas arestas desenhadas de forma a destacar, na janela de vista, que este é um componente de A. Entretanto, se A contiver uma câmera, esta ainda não é mostrada na janela de vista. Para tal, você deverá implementar o método `P2::drawCamera(c)`, em `P2,cpp`, em que `c` é uma referência para uma `Camera`. Sua implementação deverá desenhar as arestas do volume de vista da câmera em uma cor de sua escolha (que, a exemplo da cor de destaque de arestas, pode ser configurável no menu `Tools`). Uma vez que a profundidade do volume de vista pode ser muito extensa ao longo da direção de projeção, adote uma escala distinta para a profundidade, a fim de desenhar a face de trás do volume de vista mais próxima da janela de projeção. Em adição, implemente um método de *preview* da renderização da cena conforme o ponto de vista da câmera sendo desenhada. O *preview* deverá ser exibido em uma janelinha com a mesma razão de aspecto da janela de vista, com posição e altura à sua escolha. A renderização da cena deverá usar OpenGL e ser implementada no método `GLRenderer::render()`, em `GLRenderer.cpp`, conforme discutido em sala.

Além do desenho de câmeras e *preview*, implemente na sua interface gráfica um mecanismo para troca de paternidade de um objeto de cena. Por exemplo, você poderá arrastar um objeto de cena de sua posição corrente para cima do objeto de cena que será seu novo pai, ou para cima da nó da cena (nesse caso, o pai será o objeto de cena raiz da cena). Seja qual for o mecanismo que você implementar, você deverá mover um objeto de cena de qualquer ponto da hierarquia para outro. Para tal, será necessário completar os métodos com comentários `TODO` em `Transform.cpp`. Com isso, finalizaremos a implementação da classe `Transform`. Ainda na interface gráfica, implemente um mecanismo para remoção de objetos de cena na janela da hierarquia, caso ainda não o tenha. Na janela de inspeção, implemente as opções do botão `Add Component` para adição de componentes ao objeto de cena sendo inspecionado, e também a remoção de componentes (veja os comentários `TODO` no método `P2::sceneObjectGui()` em `P2.cpp`).

Observação: a partir deste exercício, não permita, se ainda não o fez, que um objeto de cena contenha mais de um componente do mesmo tipo. Portanto, um objeto de cena poderá conter somente um `Transform`, um `Primitive` e um `Camera`.

Por fim, você deverá implementar um método de foco da câmera do editor no objeto de cena correntemente selecionado: sem alterar a orientação dos eixos do VRC, a câmera deve ser posicionada a certa distância da origem do sistema local de A e o ângulo de vista ou altura da janela de projeção ajustada tal que todos os componentes de A sejam visíveis.

As atividades de programação e pesquisa dessa parte do trabalho consistem em:

A1 Implementar, em `P2.cpp`, o método `P2::drawCamera(c)`, responsável pelo desenho das arestas do volume de vista de uma câmera `c`. O método deverá ser invocado durante a renderização da cena no modo de edição (selecionado no menu `View`), sempre que o objeto de cena sendo inspecionado contiver um componente do tipo `Camera`. Para tal, use as funções de ajuste de cor de linha e de desenho de linha do editor. Considere uma escala diferente para a profundidade do volume de vista, como discutido anteriormente. Se preferir, você pode mover a definição e implementação deste método de `P2` para a classe `SceneEditor`.

A2 Implementar, em `GLRenderer.cpp`, o método `GLRenderer::render()`, responsável pela renderização da cena no modo vista de renderização (selecionado no menu `View`), usando a câmera corrente, se houver, e o mesmo programa GLSL empregado no editor.

A3 Definir e implementar, na classe `P2` ou `SceneEditor`, um método de *preview* da renderização da cena, invocado sempre que o objeto de cena sendo inspecionado

contiver um componente do tipo `Camera`. A imagem do *preview* deve ser exibida em uma janelinha definida com as funções `glViewport()` e `glScissor()`, como discutido em sala. A renderização deve usar o método implementado em **A2**.

A4 Definir e implementar, na classe `P2` ou `SceneEditor`, um método de foco da câmera do editor no objeto de cena correntemente selecionado. O método pode ser invocado, por exemplo, quando o usuário pressionar as teclas `Alt+F`.

A5 Implementar um mecanismo interativo na GUI de ajuste do pai de um objeto de cena e completar os métodos `Transform::upadte()` e `Transform::parentChanged()` em `Transform.cpp`. Além disso, limitar, no método de inspeção de `Transform` de sua GUI, os valores dos componentes de escala tais que estes não possam ser menores que um limiar positivo, por exemplo, 0.001.

A6 Implementar os métodos da GUI de criação de objetos de cena (incluindo aqueles com uma câmera), de remoção de objetos de cena e de adição e remoção de componentes do objeto de cena sendo inspecionado.

3 Passo 3: README

Como no exercício anterior, o arquivo `README` deve conter o nome(s) do(s) autor(es) e uma descrição de como gerar (caso o Visual Studio 2019 não tenha sido utilizado) e executar o programa. Em seguida, você deve descrever quais as atividades você conseguiu ou não implementar, parcial ou totalmente, além de um breve manual do usuário (por exemplo, se é preciso usar alguma tecla para alguma funcionalidade para a qual não há ajuda textual na interface gráfica com o usuário). Descreva também quaisquer outras extensões que você implementou por iniciativa própria, as quais podem valer pontuação extra em sua nota do exercício. Na descrição da atividade **A4**, apresente um resumo da matemática utilizada no cálculo do posicionamento e dimensionamento da câmera do editor.

4 Passo 4: Entrega do programa

O exercício deve ser entregue via AVA em arquivo único compactado (somente um arquivo por grupo), chamado `p2.zip` (nome(s) do(s) autor(es) vão no arquivo `README`), contendo: 1) o diretório `p2 com` o código-fonte completo e com o arquivo executável `p2.exe`, mas **sem** quaisquer arquivos intermediários de backup ou resultantes da compilação, tanto em `p2` como em seus subdiretórios (os arquivos em `/assets/meshes` também não precisam ser entregues); e 2) o arquivo `README`.

Não serão considerados e terão nota zero programas plagiados de qualquer que seja a fonte, mesmo que parcialmente, exceção feita ao código fornecido pelo professor.

5 Lista de objetivos

A verificação dos objetivos da **Parte 2** levará em conta se:

- ☐ O arquivo `p2.zip` foi submetido com os arquivos corretos, e o arquivo `README` contém as informações solicitadas.

☐ O desenho da câmera pertencente ao objeto de cena sendo inspecionado foi implementado e está correto para as projeções perspectiva (o volume de vista é um tronco de pirâmide) e paralela (o volume de vista é um paralelepípedo).

☐ A renderização da cena foi implementada na classe `GLRenderer`, e a implementação está correta (o resultado da renderização é exibido na janela gráfica no modo vista de renderização).

☐ O *preview* da renderização com a câmera corrente foi implementado, e a implementação está correta (o resultado da renderização é exibido na janelinha de *preview*).

☐ O método de foco da câmera do editor no objeto de cena correntemente selecionado foi implementado, e a implementação está correta (os componentes do objeto de cena são visíveis pela câmera do editor ao se pressionar `Alt+F`).

☐ A interface gráfica com o usuário exibe corretamente a hierarquia de objetos de cena da cena e permite a troca do pai de qualquer objeto de cena, como descrito em **A5** (após a troca de pai, as propriedades do componente `Transform` do objeto de cena são mostrados corretamente na janela `Inspector` da GUI).

☐ A criação e remoção de objetos de cena e a adição e remoção de componentes foi implementada, e a implementação está correta.