UFF – Universidade Federal Fluminense
TIC – Instituto de Computação
TCC – Departamento de Ciência da Computação

Disciplina: TCC 00.179 Computação Gráfica
Professor: Anselmo Montenegro

Trabalho de Implementação – ICGL | Divulgação: _01_/_02_/_2013_ | Entrega: _15__/_03_/_2013_

Instruções Gerais

1) Este trabalho é individual. Trabalhos entregues com código-fonte idêntico, ou com reformatações fracas (e.g., alteração de nome de variáveis, métodos ou classes) terão nota final calculada por

   [nota final] = [nota da implementação] / [número de cópias].

2) A entrega do trabalho consiste na construção de uma página na internet contendo *screenshots* dos resultados obtidos com a conclusão de cada tarefa e o envio do código fonte (apenas pastas `source` e `include`) desenvolvido pelo aluno. O código fonte não pode estar disponível na página.

3) Serão considerados entregues trabalhos enviados para o [anselmo.montenegro@gmail.com] até as 23:59 da data de entrega definida acima. Será atribuída nota ZERO a trabalhos não entregues.

4) Não é permitido ao aluno utilizar implementações ou bibliotecas compiladas de terceiros. Também é proibido ao aluno invocar comandos da API do OpenGL® ou de APIs auxiliares.

5) Leia atentamente o enunciado antes de proceder com as implementações.

Enunciado

Este trabalho consiste na implementação de procedimentos-chave que compões um *pipeline* gráfico tradicional. O trabalho está dividido em uma tarefa de aquecimento e sete tarefas de desenvolvimento com aumento gradual de dificuldade. As implementações devem ser feitas na linguagem de programação C++ e devem utilizar como ponto de partida o código fonte fornecido pelo professor. Ao término do trabalho você terá sua própria versão do ICGL (IC-UFF Graphics Library) implementada. Esta biblioteca substitui muitas das operações executadas pelo OpenGL® ou pela unidade de processamento gráfico (*graphics processing unit*, GPU) por implementações em software.

Tarefa 00 – Aquecimento

Baixe o código fonte da página do curso ou do ConexãoUFF e utilize CMake (http://www.cmake.org) para gerar um projeto ou makefile para seu compilador C++ e IDE preferidos. Na página do CMake você encontrará todas as informações necessárias para uso da ferramenta.

O código fornecido faz uso de três bibliotecas externas: DevIL (http://openil.sourceforge.net), GLUT (http://www.opengl.org/resources/libraries/glut) e GLUI (http://glui.sourceforge.net). Uma alternativa ao uso da GLUT é a freeglut (http://freeglut.sourceforge.net). Compilações das três bibliotecas para Visual Studio são fornecidas com o código fonte do projeto e podem ser encontradas na pasta `external`. Caso você utilize outra plataforma de desenvolvimento, é sua responsabilidade baixar e instalar cada uma das bibliotecas. Note que as três bibliotecas são requeridas no momento de criação do projeto ou makefile a partir do CMake.

Uma vez criado o projeto ou makefile, compile o código fonte fornecido e execute o programa. Se tudo der certo, você terá como resultado um programa gráfico que mostra dois rendering sincronizados de uma mesma cena e um painel de controle à direita. Reserve alguns minutos para explorar todos os controles disponíveis neste painel.

Inspecione os arquivos de código fornecidos e observe que dentro na pasta `include` é possível encontrar os seguintes arquivos:

`obj_loader.h`

Arquivo que contém a declaração de classes responsáveis pela carga de modelos geométricos representados no formato Wavefront's Object (OBJ) com material no formato Material Template Library (MTL).

Não é permitido alterar este arquivo!

`icgl.h`

Arquivo que contém a declaração do conjunto de operações e classes utilizadas para ensino de *pipeline* gráfico no IC-UFF. Uma descrição breve dessas operações e classes pode ser encontrada no fim deste documento.

Não é permitido alterar este arquivo!

`icgl_assignment_utils.h`

Arquivo onde você fará a declaração de funções e procedimentos úteis que poderão vir a ser utilizados em qualquer uma das oito tarefas. É garantido que tudo o que for declarado neste arquivo será visível em qualquer arquivo `.cpp` que compõe o programa.

`icgl_assignment.h`

Arquivo onde são definidas as tarefas que já foram resolvidas e a tarefa atual. Detalhes sobre a utilização deste arquivo serão dados adiante neste enunciado.

Na pasta `source` você encontrará os seguintes arquivos:

`main.cpp`

Arquivo que contém o método `main(...)` da aplicação. Ele também é responsável pela criação da janela principal, dos contextos gráficos e dos controles.

Não é permitido alterar este arquivo!

`obj_loader.cpp`

Arquivo que contém a implementação das classes responsáveis pela carga de modelos geométricos representados no formato Wavefront's Object (OBJ) com material no formato Material Template Library (MTL).

Não é permitido alterar este arquivo!

`icgl_core.cpp`

Arquivo que contém a implementação das operações e das classes que compões a ICGL.

Não é permitido alterar este arquivo!

`icgl_assignment_utils.cpp`

Arquivo onde você fará a implementação das funções e procedimentos úteis declarados por você no arquivo `icgl_assignment_utils.h`.

`icgl_assignment_{00,01,02,03,04,05,06,07}.cpp`

Arquivos onde você fará as implementações que correspondem ao cumprimento da tarefa de aquecimento e das sete tarefas principais deste trabalho.

Daqui em diante, acabou a moleza. Está na hora de colocar a mão na massa!

Para completar a Tarefa 00, abra o arquivo `icgl_assignment.h` e remova a marcação de comentário da linha contendo o seguinte código:

```
#define ICGL_ASSIGNMENT_00_WARMING_UP
```

Ao fazer isso você está dizendo para o programa que aceita realizar a parte de programação que compõe a Tarefa 00. O programa, por sua vez, para de realizar seu processamento padrão e desvia parte do processamento para as rotinas declaradas no arquivo `.cpp` correspondente à Tarefa 00. Neste caso, o arquivo `icgl_assignment_00.cpp`. Dentro deste arquivo você encontrará um procedimento incompleto:

```
void give_me_your_name(std::string &name)
```

Para completar a tarefa você terá que atribuir seu nome para o argumento de saída `name`. Por exemplo[1]:

```
name = "Ivan Sutherland";
```

Feito isso, compile o código e execute o programa. Como resultado você verá que no rendering da direita o texto “Rendered by OpenGL” será substituído pelo texto “Rendered by [seu nome] (Assignment 00)”.

Ao completar a Tarefa 00, espera-se que você tenha compreendido que para avançar na conclusão do trabalho você deverá aceitar as tarefas propostas, uma a uma. Isso é feito removendo do arquivo `icgl_assignment.h` a marcação de comentário da macro que corresponde à tarefa da vez (`ICGL_ASSIGNMENT_{número}_WARMING_UP`) e implementando as rotinas incompletas definidas no arquivo `icgl_assignment_{número}.cpp` correspondente.

Tarefa 01 – Matrizes de Rotação, Escala e Translação

Nesta tarefa você deverá completar as rotinas `make_rotation_matrix(...)`, `make_scale_matrix(...)` e `make_translation_matrix(...)` contidas no arquivo `icgl_assignment_01.cpp`, de modo que as matrizes resultantes representadas, respectivamente, pelos argumentos de saída `rotation_matrix`, `scale_matrix` e `translation_matrix` codifiquem operações de rotação, escala e translação em função dos argumentos de entrada informados. Verifique os comentários que antecedem cada uma das rotinas para obter mais informações sobre o significado de cada argumento.

É importante reforçar que, para as rotinas definidas em `icgl_assignment_01.cpp` serem invocadas pelo programa principal, é preciso remove a marcação de comentário da linha de código contendo a macro `ICGL_ASSIGNMENT_01_ROTATION_SCALE_TRANSLATION` no arquivo `icgl_assignment.h`.

A convenção adotada para construção dessas matrizes está de acordo com a especificação do OpenGL® 2.0 (veja a Seção 2.11.2 de “The OpenGL® Graphics System: A Specification”[2]).

Uma descrição breve da classe `matrix_struct`, utilizada pelas rotinas citadas acima, pode ser encontrada no fim deste documento.

A conclusão bem sucedida desta tarefa faz com que o rendering da direita na janela principal do programa seja idêntico ao resultado apresentado no rendering de referência à esquerda. Utilize os controles dos grupos “Model Transformation” e “Camera Transformation” no painel de controle para testar sua implementação.

---

[1] Veja em http://amturing.acm.org/award_winners/sutherland_3467412.cfm quem é Ivan Sutherland e quais foram suas contribuições para a Ciência da Computação, em especial para a Computação Gráfica.

[2] O document “The OpenGL® Graphics System: A Specification (Version 2.0)” está disponível em http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf.

Tarefa 02 – Matriz de Visualização

Após remover a marcação de comentário da macro `ICGL_ASSIGNMENT_02_LOOK_AT` no arquivo `icgl_assignment.h`, implemente a rotina `make_lookat_matrix(...)` definida no arquivo `icgl_assignment_02.cpp`. O comportamento esperado para esta rotina é a codificação de uma matriz de visualização no argumento de saída `lookat_matrix`. Os argumentos de entrada são explicados no próprio arquivo `.cpp`.

A matriz de visualização construída será utilizada pelo programa principal no posicionamento da câmera em cena.

Nesta tarefa é assumido que a matriz de visualização esperada é definida de acordo com a especificação da GLU® 1.3 (veja a Seção 4.1 de "The OpenGL® Graphics System Utility Library"[3]).

Tarefa 03 – Matriz de Projeção Perspectiva

Nesta tarefa você deverá construir uma matriz de projeção perspectiva a partir dos argumentos de entrada da rotina `make_perspective_matrix(...)`, definida no arquivo `icgl_assignment_03.cpp`. Esta matriz deverá ser atribuída ao argumento de saída `perspective_matrix`.

Consulte a Seção 4.1 de "The OpenGL® Graphics System: A Specification" para obter mais informações sobre o método requerido.

No arquivo `icgl_assignment.h`, remova a marcação de comentário da macro `ICGL_ASSIGNMENT_03_PERSPECTIVE_PROJECTION` a fim de ter o código implementado por você invocado pelo programa principal. Utilize os controles do grupo "Perspective Projection" para testar sua implementação.

Ao completar esta tarefa você terá substituído a implementação de todas as funções acessórias que o programa principal invocaria do OpenGL® na construção das matrizes de modelagem e visualização (*i.e.*, *modelview matrix*) e de projeção (*i.e.*, *projection matrix*). A conclusão bem sucedida das tarefas garante que as imagens renderizadas à esquerda e à direita da janela principal do programa sejam idênticas. A partir da próxima tarefa você implementará rotinas que correspondem a estágios do *pipeline* gráfico que normalmente são implementado dentro da GPU.

Tarefa 04 – Transformações Geométricas Aplicadas aos Vértices e aos Vetores Normais

O estágio de transformação e iluminação (*transformation and lighting,* T&L) do *pipeline* gráfico possui dois objetivos: (i) aplicar transformações de sistemas de coordenadas sobre vértices, normais e coordenadas de textura; e (ii) calcular a iluminação por vértice, atribuindo a este uma cor.

Na Tarefa 04 você terá que implementar a parte de transformação da T&L, enquanto que o cálculo da iluminação por vértice será resolvido na próxima tarefa. Para aceitar a tarefa atual, defina a macro `ICGL_ASSIGNMENT_04_VERTEX_TRANSFORMATION` no arquivo `icgl_assignment.h` e implemente a rotina `vertex_transformation(...)` que se encontra incompleta no arquivo `icgl_assignment_04.cpp`.

Ao aplicar transformações geométricas, as coordenadas dos vértices do modelo geométrico são mapeadas do sistema de coordenadas de objeto (*object coordinates*, OC) para o sistema de coordenadas da câmera (*eye coordinates*, EC) e, em seguida, para o sistema de coordenadas de recorte (*clip coordinates*, CC). Neste estágio, os vetores normais associados aos vértices do modelo também sofrem transformações que os mapeiam de OC para EC e, por fim, os escalam para norma igual a 1.

[3] O documento "The OpenGL® Graphics System Utility Library (Version 1.3)" está disponível em http://www.opengl.org/registry/doc/glu1.3.pdf. Informações adicionais podem ser obtidas na Web.

Identifique no protótipo da rotina `vertex_transformation(...)` e nos comentários que a antecedem no arquivo `.cpp` quais são os argumentos de entrada e quais são os argumentos de saída que você deverá calcular. Utilize o material visto em aula e as Seções 2.11.1 e 2.11.3 de "The OpenGL® Graphics System: A Specification"[2] como fonte de consulta para a implementação da solução.

Descrições em alto-nível das classes `location_struct` e `direction_struct` utilizadas pela rotina `vertex_transformation(...)` são apresentadas no fim deste documento.

Ao concluir esta tarefa você terá um resultado semelhante ao apresentado no rendering de referência, porém com intensidades diferentes de cor, pois a iluminação ainda não foi propriamente resolvida. O importante ao término da Tarefa 04 é verificar se a localização dos vértices e se os vetores normais do rendering da ICGL são idênticos aos do rendering de referência. Utilize as opções "Display normal vectors" e "Wireframe mode enabled", disponíveis no painel de controles à direita, para auxiliar a depuração.

Tarefa 05 – Cálculo da Cor dos Vértices

Ao remover a marcação de comentário da macro `ICGL_ASSIGNMENT_05_VERTEX_LIGHTING` no arquivo `icgl_assignment.h`, você estará instruindo o programa a utilizar o seu cálculo de iluminação por vértice ao invés do cálculo realizado pela GPU. Entretanto, o resultado obtido por você ao implementar a rotina `vertex_lighting(...)` contida no arquivo `icgl_assignment_05.cpp` deverá ser idêntico ao exibido pelo rendering de referência na janela principal do programa.

Logo, a Tarefa 05 consiste em resolver a parte de iluminação da T&L, aplicando o modelo de reflexão de Blinn-Phong[4] sobre o vértice processado atualmente pela rotina `vertex_lighting(...)` e atribuindo a cor resultante ao argumento de saída `vertex_color` caso o argumento de entrada `lighting_enabled` seja igual a `true`. Caso contrário, `vertex_color` recebe a cor definida pelo argumento de entrada `base_color`. Utilize os controles do grupo "Light" no painel de controle para testar sua implementação.

A operação de iluminação do ICGL é similar à realizada pelo OpenGL®, cuja descrição detalhada pode ser encontrada na Seção 2.14.1 de "The OpenGL® Graphics System: A Specification"[2].

Uma descrição breve da classe `color_struct`, utilizada pela rotina `vertex_lighting(...)` é apresentada no fim deste documento.

Tarefa 06 – Montagem de Primitivas, Recorte e Supressão de Faces Traseiras

Nas duas tarefas anteriores, o processamento executado pelas rotinas teve como entrada os atributos de um único vértice e a saída foi o cálculo de outros atributos para esse mesmo vértice. Na tarefa atual, todos os vértices processados individualmente em etapas anteriores serão informados na forma de listas de atributos que deverão ser agrupados em primitivas, podendo essas ser: (i) segmentos de reta; ou (ii) triângulos.

Quando as primitivas em questão forem segmentos de reta, a rotina `make_segments(...)` será invocada pelo programa. No caso de triângulos, a rotina `make_triangulos(...)` será invocada. Seu trabalho aqui é fazer a implementação de ambas as rotinas no arquivo `icgl_assignment_06.cpp` e permitir que elas sejam utilizadas pelo *pipeline* gráfico ao definir a macro `ICGL_ASSIGNMENT_06_PRIMITIVE_ASSEMBLY_CLIPPING_AND_CULLING` no arquivo `icgl_assignment.h`.

Para a implementação dessas rotinas é importante notar que segmentos de reta são definidos por dois vértices e que triângulos por três. A convenção adotada neste trabalho é que pares (ou trios) consecutivos de vértices nas listas de atributos definem uma primitiva. Logo, em uma situação em que

[4] O modelo de reflexão de Blinn-Phong foi publicado no artigo "J. F. Blinn. Models of light reflection for computer synthesized pictures. Proceedings of SIGGRAPH, 1977, pp. 192-198", disponível em http://dx.doi.org/10.1145/965141.563893.

todos os N vértices gerem primitivas a serem rasterizadas na próxima tarefa, o número de primitivas geradas no caso de segmentos será $N/2$ e no caso de triângulos $N/3$.

Entretanto, é pouco provável que todas as primitivas possíveis sejam geradas, pois muitas delas podem ser descartadas do processo de rasterização por estarem fora do campo de visão da câmera (*frustum*). A esta operação é dado o nome de recorte (ou *clipping*). Além disso, caso a supressão de faces traseiras esteja habilitada, triângulos cuja projeção possua orientação oposta à assumida para uma face frontal devem ser suprimidas (*backface culling*). Este cuidado permite que a etapa posterior (*i.e.*, rasterização) tenha seu curso computacional reduzido enormemente.

Logo, antes de agrupar as propriedades dos vértices e de inseri-los como primitivas no argumento de saída `primitives`, é preciso realizar a operação de recorte e a operação de supressão de faces traseiras, caso esta esteja habilitada. Consulte os comentários que antecedem as rotinas `make_segments(...)` e `make_triangles(...)` para verificar quais atributos de vértice são fornecidos, como é definida a orientação padrão para faces frontais e como a supressão de faces traseiras é indicada como habilitada ou desabilitada. Para facilitar a implementação, considere que qualquer primitiva que tenha ao menos um vértice fora do campo de visão da câmera deve ser removida por completo, sem a necessidade de um recorte preciso da porção que poderia estar sendo visualizada. Por conta desta simplificação, é de se esperar que o resultado visual obtido pelo rendering com ICGL será diferente do obtido com OpenGL® caso alguma primitiva seja parcialmente visível pela câmera.

No fim deste documento você encontrará uma descrição breve dos elementos que compõe as classes `texcoord_struct`, `segment_struct` e `triangle_struct`, utilizadas pelas rotinas de interesse da tarefa atual. Note que nas classes de primitivas existe a necessidade de armazenar as coordenadas dos vértices em CC. Consulte o material de aula e a Seção 2.12 de "The OpenGL® Graphics System: A Specification"[2] para verificar como o sistema CC é utilizado na operação de recorte.

Tarefa 07 – Rasterização, Aplicação de Textura e Teste de Profundidade

A última tarefa consiste em tomar uma primitiva (*i.e.*, segmento de reta ou triângulo) que sobreviveu ao recorte e a supressão de faces traseiras (no caso de triângulo), mapear seus vértices de CC para coordenadas normalizadas de dispositivo (*normalized device coordinates*, NDC) e em seguida para coordenadas de janela (*window coordinates*, WC). Uma vez em WC, é preciso converter a primitiva contínua em fragmentos discretos (processo de rasterização) e, durante este procedimento, atribuindo a cada fragmento uma cor e uma profundidade. Os fragmentos, por sua vez, são utilizados na atualização do buffer profundidade (*depthbuffer*) caso o teste de profundidade (*depth teste*) esteja habilitado, e do buffer de cor (*framebuffer*) caso o fragmento tenha passado no teste de profundidade.

A cor do fragmento deverá ser obtida por interpolação linear da cor dos vértices da primitiva e, no caso do mapeamento de textura estar habilitado, os valores amostrados da textura deverão entrar na composição da cor final. Veja as Seções 3.4.1 e 3.51 de "The OpenGL® Graphics System: A Specification"[2] para obter uma explicação simples de como a rasterização e interpolação podem ser feitas a partir de CC para, respectivamente, segmentos de reta e triângulos. A Seção 2.11 explica como NDC e WC são obtidas para cada vértice.

As operações de rasterização, aplicação de textura e teste de profundidade deverão ser implementadas nas rotinas `rasterize_segment(...)` e `rasterize_triangle(...)` definidas no arquivo `icgl_assignment_07.cpp`. Essas rotinas tornam-se ativas no programa a partir da definição da macro `ICGL_ASSIGNMENT_07_RASTERIZATION` no arquivo `icgl_assignment.h`. Leita os comentários que antecedem as rotinas para obter mais informações sobre seus argumentos de estrada e saída.

Bom Trabalho!

Classes Disponíveis na ICGL

`color_struct`

Codifica uma cor no sistema de cores RGBA. Cada um dos canais de cor pode assumir valores reais no intervalo $[0,1]$. O valor armazenado por um canal específico pode ser acessado por meio dos atributos `r` (vermelho), `g` (verde), `b` (azul) e `a` (alpha) da classe, ou por indexação direta como é feito com arrays (*e.g.*, `aux[0]`, `aux[1]`, `aux[2]` e `aux[3]`, onde `aux` é uma variável do tipo `color_struct`).

`direction_struct`

Codifica uma direção em um espaço cartesiano tridimensional. Cada uma das coordenadas do vetor de direção pode assumir valores reais no intervalo $(-\infty, +\infty)$. O valor armazenado por uma coordenada específica pode ser acessado por meio dos atributos `x`, `y` e `z` da classe, ou por indexação direta como é feito com arrays (*e.g.*, `aux[0]`, `aux[1]` e `aux[2]`, onde `aux` é uma variável do tipo `direction_struct`).

`location_struct`

Codifica a localização de um ponto em um espaço homogêneo tridimensional. Cada uma das coordenadas do ponto pode assumir valores reais no intervalo $(-\infty, +\infty)$. O valor armazenado por uma coordenada específica pode ser acessado por meio dos atributos `x`, `y`, `z` e `w` da classe, ou por indexação direta como é feito com arrays (*e.g.*, `aux[0]`, `aux[1]`, `aux[2]` e `aux[3]`, onde `aux` é uma variável do tipo `location_struct`).

`matrix_struct`

Representa uma matriz 4×4, cujas células estão organizadas na memória conforme o padrão adotado pelo OpenGL®, ou seja, alinhados por coluna (veja a Seção 2.11.2 de "The OpenGL® Graphics System: A Specification"[2]). O acesso a uma célula pode ser feita por indexação simples como é feito com arrays (*e.g.*, `aux[0]`, `aux[1]`, ..., `aux[15]`, onde `aux` é uma variável do tipo `matrix_struct`), ou por indexação dupla (*e.g.*, `aux(r,c)`, onde `aux` é uma variável do tipo `matrix_struct` e `r` e `c` são, respectivamente, os índices de linha e coluna da célula).

`segment_struct`

Classe que agrega todos os atributos associados aos vértices de um segmento de reta, dentre eles, a quantidade de vértices (`vertices_count`), a localização dos mesmos em CC (`vertex_cc`), as cores calculadas durante a etapa de T&L (`color`), as coordenadas de textura (`texcoord`) e o identificador da textura mapeada sobre esta primitiva (`texture_id`).

`texcoord_struct`

Codifica coordenadas de textura bidimensional. Cada uma das coordenadas pode assumir valores reais no intervalo $[0,1]$. O valor armazenado por uma coordenada específica pode ser acessado por meio dos atributos `u` e `v` da classe, ou por indexação direta como é feito com arrays (*e.g.*, `aux[0]` e `aux[1]`, onde `aux` é uma variável do tipo `texcoord_struct`).

`triangle_struct`

Classe que agrega todos os atributos associados aos vértices de um triângulo, dentre eles, a quantidade de vértices (`vertices_count`), a localização dos mesmos em CC (`vertex_cc`), as cores calculadas durante a etapa de T&L (`color`), as coordenadas de textura (`texcoord`) e o identificador da textura mapeada sobre esta primitiva (`texture_id`).