UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO
DEPARTAMENTO DE ELETRÔNICA E COMPUTAÇÃO

DRAWTEX SCV: UMA FERRAMENTA EDUCATIVA PARA A COMPOSIÇÃO DE PRIMITIVAS GRÁFICAS OPENGL COM EXPORTAÇÃO DE CÓDIGO C/C++

TRABALHO FINAL DE GRADUAÇÃO

Vinícius Michel Gottin

Santa Maria
2010

DRAWTEX SCV: UMA FERRAMENTA EDUCATIVA PARA A COMPOSIÇÃO DE PRIMITIVAS GRÁFICAS OPENGL COM EXPORTAÇÃO DE CÓDIGO C/C++

por

Vinícius Michel Gottin

Trabalho Final de Graduação apresentado ao Curso de Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação.

Orientador Prof. Dr. Cesar Tadeu Pozzer

Trabalho de Graduação N. 317
Santa Maria, RS, Brasil
2010
Universidade Federal de Santa Maria

Centro de Tecnologia
Curso de Ciência da Computação
Departamento de Eletrônica e Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho Final de Graduação

DRAWTEX SCV: UMA FERRAMENTA EDUCATIVA PARA A COMPOSIÇÃO DE PRIMITIVAS GRÁFICAS OPENGL COM EXPORTAÇÃO DE CÓDIGO C/C++

elaborado por

Vinícius Michel Gottin

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

_____

Dr. Cesar Tadeu Pozzer- UFSM
(Presidente/Orientador)

_____

Dra. Márcia Pasin - UFSM

_____

Msc.Fernando Bevilacqua–UFFS-Chapecó

Santa Maria, 2010.

Às pessoas importantes na minha vida.

AGRADECIMENTOS

Agradeço aos colegas que proporcionaram horas de diversão em meio às horas de calmaria aparente seguidas por terror acadêmico puro, especialmente àqueles que compartilharam projetos, idéias, falhas e sucessos em algum dos grupos dos quais participei, desde o diretório acadêmico até em laboratório de pesquisa. Também agradeço à Janice Vendruscolo pela ajuda em todas as dúvidas que tive sobre a vida acadêmica, potencializadas pelo fato de não ter seguido a ordem recomendada do curso.

Aos professores que proporcionaram horas de terror acadêmico aparente, mas que resultaram em muito aprendizado - e algumas noites não-dormidas. Agradeço em especial aos professores que contribuíram diretamente com minha graduação, possibilitando que eu realizasse diversas tarefas durante esse período: a professora Andrea Charão, pelo apoio e orientação no grupo PET; aos professores Raul Ceretta, Antonio Marcos de Oliveira Candia e Benhur Stein, coordenadores do curso durante este período e que, mais de uma vez me ajudaram com problemas burocráticos; aos professores Marcos D'Ornellas e Lisandra, pela convivência no Laboratório de Computação Aplicada e as oportunidades que ali surgiram; e principalmente ao professor Cesar Pozzer, pela orientação e ajuda que extrapolou os projetos de pesquisa.

Agradeço também aos meus pais e irmão, pelo suporte e paciência com minha segunda faculdade. Vocês são os principais responsáveis pelo que eu sou e tudo que fiz. Finalmente, agradeço à Luiza, pois sem seu apoio e companheirismo eu não teria aproveitado nenhum dos sucessos que obtive durante a Ciência da Computação. Eu teria, provavelmente, me alistado na Legião Estrangeira ou em programas secretos do governo norte-americano para viajar no tempo e matar Hitler.

“http://www.youtube.com/watch?v=7CsNZp5lsCI”

RESUMO

Trabalho de Graduação

Curso de Ciência da Computação

Universidade Federal de Santa Maria

DRAWTEX SCV: UMA FERRAMENTA EDUCATIVA E PARA A COMPOSIÇÃO DE PRIMITIVAS GRÁFICAS OPENGL COM EXPORTAÇÃO DE CÓDIGO C/C++

AUTOR: VINÍCIUS MICHEL GOTTIN

ORIENTADOR: DR. CESAR TADEU POZZER

Data e Local da Defesa: 10 de Dezembro de 2010, Santa Maria

Considerando-se a dificuldade do programador OpenGL aprendiz em compreender o efeito das diversas variáveis e estados internos da API no resultado final de uma renderização, bem como a dificuldade de compor objetos tridimensionais em código a partir de chamadas de desenho do OpenGL; o objetivo deste projeto é produzir uma ferramenta especificamente com o propósito de permitir a composição de primitivas gráficas em um ambiente que proporcione o aprendizado necessário sobre o OpenGL. Assim, para sanar estas dificuldades, objetiva-se a inclusão de um módulo de exportação de código C/C++ a partir de uma interface de edição e visualização parametrizável de vértices e primitivas em espaço 3D. A implementação seguiu um padrão incremental, com um ciclo de prototipação, o que permite a modularização e fácil expansão da ferramenta, e utilizou-se da API SCV para a composição da interface gráfica; tendo em vista a sua validação em aplicativos maiores que trabalhos acadêmicos. O software produzido foi chamado DrawtexSCV e satisfaz os requisitos propostos, contando ainda com várias possibildades de expansão já encaminhadas.

Palavras-chave: OpenGL, SCV, computação gráfica, composição, ferramenta educativa

ABSTRACT

Undergraduate Final Work

Graduation in Computer Science

Federal University of Santa Maria

DRAWTEX SCV: AN OPENGL PRIMITIVES COMPOSITION EDUCATIONAL TOOL WITH C/C++ CODE EXPORTATION FEATURE

AUTHOR: VINÍCIUS MICHEL GOTTIN

ADVISOR: CESAR TADEU POZZER, DR.

Date and Local: December10 2010, Santa Maria

Considering the difficulty of learning, as an OpenGL programmer, the effect of the several internal state variables of this API in a rendered scene, as well as the difficulty of composing three-dimensional objects in code description of OpenGL calls; this project aims to produce an educational tool to allow the composition of graphic primitives in an educational environment that focuses on concepts of OpenGL. Thus, in order to step over these difficulties, the objective of the present work is to produce a C/C++ code exportation feature from an editing and visualization interface of 3D vertexes and primitives. The project followed and incremental pattern, counting with the building of a prototype before the final implementation started. This allowed for modularization and ease of future expansion. The SCV API for the composition of GUIs was applied, in order to validate its use in more complex software. The results satisfy the proposed objectives, and the project presents various possibilities of expansion.

Keywords: *OpenGL, SCV, computer graphics, composition, educational tool*

.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

2D - bidimensional

3D – tridimensional

API – *Application Programming Interface*

ARB - *Architecture Review Board*

CAD – *Computer Aided Design*

GLUT - *OpenGL Utility Toolkit*

GUI – *Graphic User Interface*

OpenGL– *Open Graphic Library*

SCV – *SimpleComponents for Visual*

SGI - *Silicon Graphics Incorporated*

SUMÁRIO

1 INTRODUÇÃO

A Computação Gráfica é um assunto de grande relevância para a ciência e indústria devido a grande gama de aplicações que se beneficiam com a exibição de dados em formato gráfico. Os campos específicos desta área incluem PresentationGraphics (gráficos de apresentação), Computer art (arte computacional), Entertainment (entretenimento), Educationand Training (educação e treinamento), Visualization (visualização), ImageProcessing (processamento de imagens), Graphical User Interfaces (GUI – interfaces gráficas de usuário) e Computer Aided Design (CAD – Desenho assistido por computador)[4].

Vários destes campos promovem diferentes aplicações de modelos tridimensionais, representações matemáticas descritivas de superfícies e objetos em um espaço tridimensional. Os usos de modelos 3D são pertencentes a dois tipos, primariamente: **exposição**, onde – geralmente – são transformados em imagens bidimensionais a partir de um processo conhecido como *rendering*; e **simulação**, onde, a partir de alguma perturbação, os dados descritivos do modelo configuram o próprio resultado. Independente do uso, no entanto, modelos tridimensionais costumam ser originados a partir de softwares complexos, baseados em alguma das bibliotecas gráficas em uso na atualidade.

Bibliotecas gráficas são interfaces de software para hardware gráfico, permitindo a portabilidade de código de programação entre diferentes arquiteturas de processadores e placas gráficas e oferecendo aos programados conjuntos de operações pré-definidas para a manipulação deste hardware. Uma das principais bibliotecas gráficas em uso na atualidade é o padrão aberto da indústria de placas gráficas aceleradoras [11**Error! Reference source not found.**]; o OpenGL, é parte do tema deste trabalho: a produção de um software simples de modelagem que permita a produção de primitivas gráficas de forma visual e interativa.

A motivação para este trabalho vem do fato de que há uma grande dificuldade em obter modelos tridimensionais complexos para aplicações em OpenGL, por diversos motivos; especialmente para o programador iniciante, que enfrenta dificuldades no aprendizado do funcionamento da biblioteca devido à sua enorme complexidade.

Visando satisfazer as necessidades de estudantes de Computação Gráfica de nível superior, este trabalho se propõe a produzir uma ferramenta educativa, que favoreça a compreensão do funcionamento do OpenGL em sua execução, com o objetivo prático de possibilitar a composição de primitivas gráficas do OpenGL, até mesmo combinadas em modelos simples, para a exportação de código das mesmas em C/C++.

1.1 Objetivos

O objetivo principal do trabalho é produção de uma ferramenta especializada para a facilitação de descrição de objetos tridimensionais em código através da composição e visualização em interface interativa tridimensional, contando ainda comexportação de código parametrizável. A visualização deve atender aos propósitos educativos da ferramenta, possibilitando que o usuário perceba graficamente o resultado das diferentes parametrizações das variáveis e estados do OpenGL.

Diversos objetivos específicos de implementação também são definidos; figurando entre eles a organização do software de forma a permitir fácil expansão futura e a análise e validação do uso da API SCV para construção de interfaces gráficas em programas de âmbito maior que o de trabalhos acadêmicos.

1.2 Organização do texto

O capítulo 2, Revisão bibliográfica e fundamentação, apresenta uma revisão de conceitos necessários para a realização do trabalho. Os assuntos abordados são, principalmente, características de alto nível da biblioteca gráfica OpenGL e da API SCV. No capítulo 3, Objetivo do software e planejamento, é dada a motivação para a realização do trabalho, e são estabelecidos em detalhe os objetivos geral e especifícos da ferramenta. Além disso, é discutido o planejamento e implementação do protótipo da ferramenta, denominado *Drawtex*.

Já no capítulo 4, DRAWTEX SCV, é discutida a implementação final da ferramenta, com especial atenção à estruturação do programa e algumas classes que definem processos importantes relacionados aos objetivos do trabalho. Ao longo do texto, principalmente deste últim capítulo, a mudança de fontes indica itens de código: nomes **`destacados`** referem-se a atributos, nomes de métodos e instâncias; nomes destacados **VERSALETE** à nomes de classes.

2 REVISÃO BIBLIOGRÁFICA E FUNDAMENTAÇÃO

2.1 OpenGL

A biblioteca gráfica OpenGL é um dos padrões da indústria de hardware gráfico. As origens deste padrão remontam à biblioteca ***IRIS GL*** [12] da ***SGI*** (*Silicon Graphics Incorporated),* criada para as estações da linha *IRIS,* portadoras de hardware especializado e otimizado para a renderização de gráficos sofisticados para a época. A IRIS GL(conhecida também apenas como *GL*) era, portanto, proprietária e intimamente ligada a uma arquitetura de hardware específica. O OpenGL, no entanto, apesar de baseado naquela, foi lançado em 1992 pela SGI como especificação de domínio público [www.opengl.org] e desde então vem sendo mantido pela ARB (*Architecture Review Board*), um conselho formado por empresas de tecnologia ligadas à produção de hardware e softwares gráficos [6]. O OpenGL é foco deste trabalho por suas características de padrão aberto da indústria e amplo uso na comunidade acadêmica.

Com a inexistência, no OpenGL, de um sistema de gerenciamento de janelas próprio (consequência indireta de sua necessidade de portabilidade), é necessária a utilização de uma biblioteca auxiliar para a integração entre a aplicação produzida e o sistema de janelas do sistema operacional. Este papel é geralmente desempenhado pela GLUT (*OpenGL Utility Toolkit*), em parte pela sua simplicidade, e em parte por ser especialmente apropriada para o aprendizado do funcionamento do OpenGL. Assim como o OpenGL, a GLUT foi utilizada neste trabalho e algumas de suas características específicas são abordadas ao longo do texto, quando necessário esclarecimento, especialmente no capítulo 4, DRAWTEX SCV. Uma breve visão sobre alguns tópicos de alto nível necessários para a compreensão do trabalho, como a arquitetura geral da API OpenGL, a representação de primitivas, o fluxo de renderização e a estruturação geral do OpenGL são dadas nas próximas seções.

2.1.1 Arquitetura da API OpenGL

Um dos propósitos principais da adoção de um padrão de bibliotecas gráficas é a portabilidade entre diferentes conjuntos de hardware. Assim, uma aplicação construída sobre o OpenGL não se comunica diretamente com a placa gráfica, mas

apenas com a API do OpenGL. Esta API oferece mais de 250 comandos ao programador, para que ele especifique objetos e operações referentes a um espaço tridimensional (ou bidimensional, quando necessário, como será visto na seção 2.1.2, a seguir). Uma representação desta API como interface entre a aplicação e o hardware gráfico pode ser vista na Figura 1.

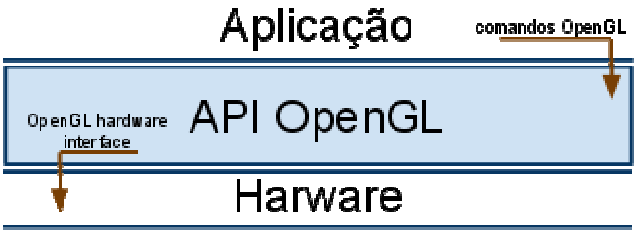

Figura 1 Abstração gráfica do OpenGL como API e interface entre aplicação e hardware gráfico.

Nota-se que, independente da implementação do OpenGL da qual dispõe o programador, é papel da API transformar todos os comandos do nível de aplicação necessários em comandos de hardware. Também é papel do OpenGL, em caso de inexistência de placa gráfica, suportar as funcionalidades não oferecidas, atuando como um *driver* virtual que executa na CPU [9].

Os comandos disponibilizados ao programador modificam o contexto do OpenGL, que se comporta como uma máquina de estados. A cor atual, por exemplo, é uma variável de estado: tendo setado a cor para branco, vermelho ou qualquer outra, todos os objetos desenhados subsequentemente serão dessa cor, até que uma nova chamada troque a cor atual. Todas as variáveis de estado, ou modos, tem um valor padrão; e são disponibilizadas chamadas que retornam o valor atual dessas variáveis, com o uso das quais o programador pode descobrir, por exemplo, qual é a cor atual no estado do OpenGL.

2.1.2 Representação gráfica de primitivas OpenGL

Como o propósito principal do OpenGL é oferecer um meio de produzir representações gráficas de dados, encontra-se dentre os comandos da API básica um rol de comandos para definir modelos tridimensionais. Estes modelos (também chamados de *objetos*) são construídos a partir de primitivas geométricas – pontos, linhas e polígonos - especificados por seus vértices. Assim, o programador OpenGL preocupa-se em descrever vértices, no espaço, que formam linhas ou polígonos, e a API trata de relacioná-los uniformemente ao longo de seu processo de renderização. Ou seja, as primitivas oferecidas ao programador para descrever seus objetos são,

invariavelmente, resumidas em algum ponto do processo de renderização a seus vértices e como eles se relacionam entre si.

Um vértice, na concepção do OpenGL, não possui dimensões – é apenas uma abstração matemática que define um ponto no espaço. Quando o OpenGL calcula os resultados da rasterização, ele é impossibilitado de representar um ponto como algo menor que um pixel – e o resultado típico é, portanto, desenhá-lo como um único pixel. Assim, muitos pontos com coordenadas similares em espaço de coordenada podem ser representados pelo mesmo pixel na tela.

O ponto sempre é considerado pela implementação do OpenGL como pertencente a um espaço tridimensional, embora o programador possa ignorar a terceira coordenada (*Z*), que tem por padrão o valor zero. Assim, em uma aplicação que o programador deseja ser bidimensional, todos os pontos (e todas as primitivas) estão contidas no plano *XY.*

Uma definição de um polígono no OpenGL pode ser feitas de várias maneiras, mas o modo de desenho mais básico é chamado de *modo imediato*. Neste modo, o programador inicia o processo ao realizar uma chamada *glBegin*, passando como parâmetro o tipo de primitiva que quer formar, e o termina com uma chamada *glEnd*. Entre essas duas chamadas, realiza diversas chamadas *glVertex*, passando como parâmetros as posições de cada vértice no espaço. A Figura 2 mostra os resultados obtidos com cada uma das primitivas disponibilizadas pelo OpenGL, e os números representam a ordem em que os vértices foram determinados.

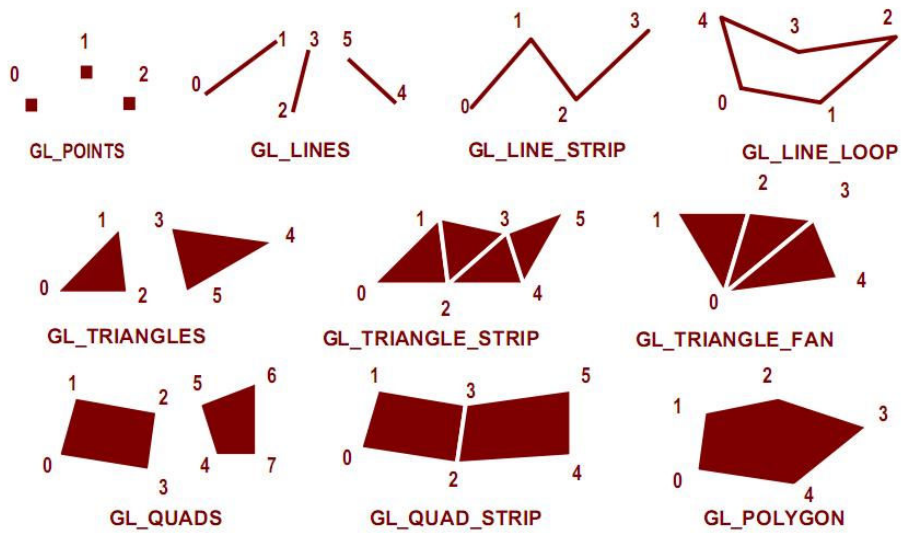

Figura 2 Primitivas OpenGL. [9]

Os vértices são aplicados pelo OpenGL na ordem em que são definidos para formar a primitiva desejada (aquela que foi passada como parâmetro na chamada *glBegin*). A Figura 3 mostra um trecho de código e o resultado obtido com sua execução, e o efeito da troca de ordem de dois vértices no resultado final de uma chamada *GL_TRIANGLES* (com a coordenada *Z* desconsiderada). Na figura, os vértices 0 e 2 da porção superior são invertidos na porção inferior. Os triângulos gerados (na esquerda) e o código gerador correspondente (na direita) estão coloridos para identificação, sendo que a numeração em ambas as representações refere-se aos vértices, em ordem, do código na porção superior da figura. A ordem dos vértices também influencia na definição da direção das faces (*front face*, *back face*), utilizada pelos algoritmos de *face culling* e definição de aplicação de *materials*. Por esses exemplos simples, pode-se compreender a dificuldade na definição manual de simples primitivas gráficas comumente usadas em aplicações opengl.

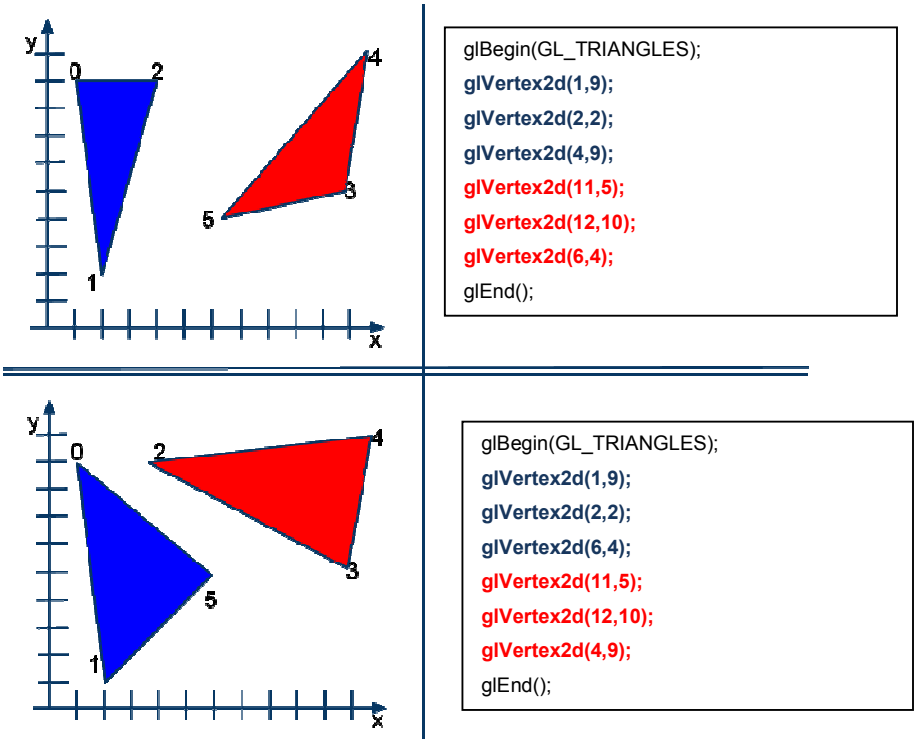

Figura 3 Resultado da troca de ordem de dois vértices em uma sequência de chamadas *glVertex* com a primitiva *GL_TRIANGLES* ativada.

2.1.3 O fluxo de renderização OpenGL

Até a versão 3.0, o OpenGL contava com um fluxo de operações internas. Este fluxo, referido como *rendering pipeline*, era composto de uma série de estágios fixos de processamento. A partir da versão 3.1 esta série tornou-se descontinuada oficialmente, embora ainda suportada a partir de extensões naturalmente inclusas nas distribuições do OpenGL. Versões seguintes (atualmente em fase de transição para a 4.0) aplicam mudanças ainda mais profundas, mas de resultado equivalente e, até certo ponto, retrocompatíveis. No que diz respeito ao presente trabalho estas mudanças não são importantes, mas alguns conceitos implementados fazem uso da relação entre essas operações e, portanto, é útil compreender o *pipeline* do OpenGL conforme sua concepção usual. Este *pipeline* é representado graficamente na Figura 4.

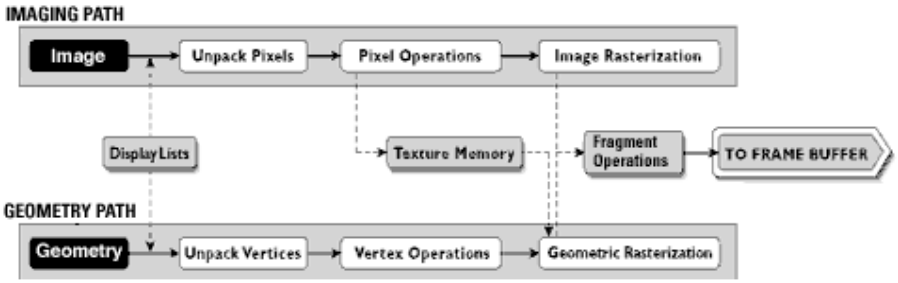

Figura 4 Fluxo de operações [fonte: www.opengl.org]

Nota-se pela análise da figura que a informação geométrica segue um caminho que inclui **operações de vértice** (*Vertex Operations)* e **rasterização geométrica**(*Geometric Rasterization*). Já a informação de fragmento (*Imaging Path*) passa pelas etapas de **operação por pixel** (*Pixel Operations*) e **rasterização de imagem** (*ImageRasterization*).Ambos os caminhos se unem no passo de **operações por fragmento** (*Fragment Operations*), e culminam na informação sendo guardada no *frame buffer*.

Antes de explorar o conceito de *framebuffer* é interessante lembrar que o resultado de cada iteração desse *pipeline* é informação de pixels. Ou seja, o *frame buffer* do OpenGL é o conjunto de *buffers,* cada um deles sendo uma matriz retangular onde é guardada um dado para cada pixel (esta memória é localizada, geralmente, na placa gráfica). Assim, um *ColorBuffer*, por exemplo, é uma matriz onde é guardadaum dado de cor para cada pixel. Outro *buffer* do OpenGL de interesse para este trabalho é o *DepthBuffer,* que guarda a informação de profundidade utilizada para cálculo de oclusão de objetos.

Assim, o *frame buffer* é onde o OpenGL guarda as informações resultantes do processo de *pipeline*, e pode ser entendido como uma matriz retangular onde é guardada toda a informação para cada pixel. É a partir da informação no *frame buffer* que se torna possível, para um computador, representar informação graficamente em um monitor.

Em aplicações mais simples, a ordem direta do pipeline é preservada sem desvios. Funções mais avançadas não raro exigem, no entanto, que sejam lidos dados de algum *buffer* específico. Isto foi abordado, neste trabalho, nas funções de *picking*, conforme visto na seção 4.2.3.1.

2.1.4 Estrutura de programa OpenGL

Programas interativos em OpenGL utilizando a GLUT para gerenciamentode janelas geralmente funcionam como um laço infinito, durante o qual o OpenGL controla o fluxo de execução, aplicando as funções determinadas pelo programador. Essas funções, referenciadas como *callbacks*, são definidas na GLUT e incluem: função de entrada de dados de mouse, de entrada de dados de teclado, de plano de fundo; e várias outras (como *callbacks* de janela e menu). Este conceito é apresentado na Figura 5.

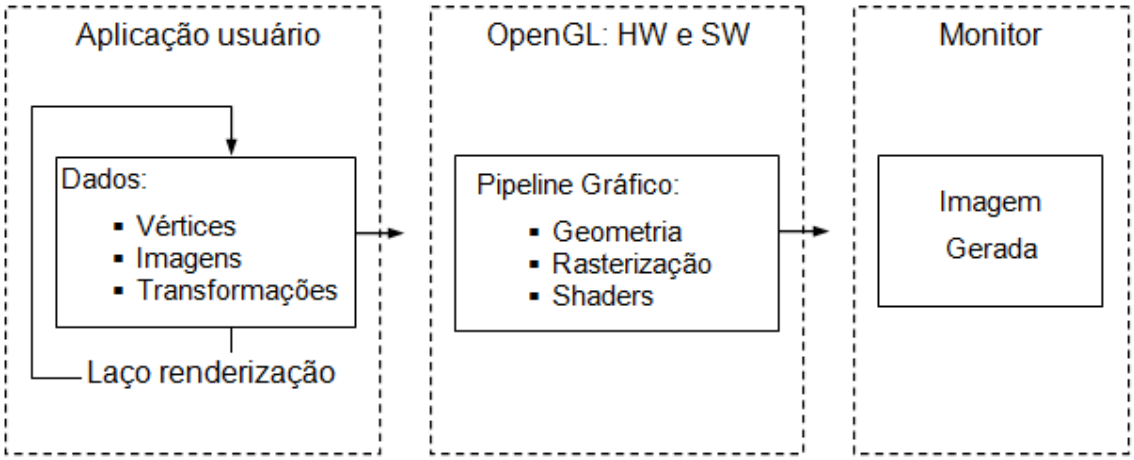

Figura 5 Estrutura da relação entre a aplicação do usuário e o OpenGL, mostrando o fluxo de processos até a renderização em tela [9].

O programador define as *callbacks* com as chamadas GLUT específicas, e depois ativa o laço de execução infinito com uma chamada *glutMainLoop*. Todas as *callbacks* definidas são então, durante este laço, chamadas quando os eventos correspondentes acontecem. A *callback* de mouse, por exemplo, é definida com uma chamada à função *glutMouseFunc* (passando como argumento um ponteiro para uma função com a assinatura específica). Se o programador cria, em seu código, a função *testFunc* (com a assinatura correta) e chama *glMouseFunc* passando um ponteiro para *testFunc* como parâmetro, a GLUT trata de, durante o laço de execução, executar a função *testFunc* toda vez que for detectado um evento de mouse.

O software produzido neste trabalho faz uso de uma API adicional, chamada SCV, construída sobre a GLUT, de tal forma que as *callbacks* nele presente se adequam ao modelo do SCV, e não da GLUT. As distinções entre esses modelos podem ser vistas na seção 2.2, a seguir.

2.2 O SCV

A construção de interfaces gráficas de usuário (GUIs) avançadas é um tópico complexo, e as funcionalidades para tal da GLUT são muito primitivas e fazem com que, não raro, seja necessário mais tempo para programar a interface do que as funcionalidades do programa - este problema pode ser detectado especialmente em trabalhos acadêmicos que envolvem a programação de aplicativos gráficos. Com o intuito de satisfazer as necessidades imediatas de alunos de Ciência da Computação da Universidade Federal de Santa Maria (UFSM) nas disciplinas em que produzem software que necessitam de interfaces gráficas – principalmente aquelas ligadas a computação gráfica -, foi desenvolvida uma API denominada "SimpleComponents for Visual" (SCV) [12], desenvolvido com base na biblioteca gráfica *OpenGL,* utilizando a linguagem de programação C++. As primeiras versões foram construídas ainda sobre a GLUT; a versão atual é baseada na freeglut [http://freeglut.sourceforge.net/], uma alternativa de código aberto da GLUT.

A principal característica pretendida da API, em sua concepção original, era oferecer ao programador as ferramentas necessárias para a composição de GUIs atrativas e funcionais, similares às interfaces de softwares comerciais reconhecidos; com foco na simplicidade de uso, de forma que ele pudesse utilizar-se da API de forma intuitiva na composição de GUI em seu software. A API SCV também deveria ser portável para variadas IDEs (ambientes de desenvolvimento, *IntegratedDevelopmentEnvironment*), e os sistemas operacionais *Windows* e *Linux*.

As primeiras versões foram disponibilizadas a alunos, conforme seu propósito original, e verificou-se a sua validação com o uso extensivo em trabalhos acadêmicos. O projeto de desenvolvimento do SCV prosseguiu, utilizando-se da análise dos resultados obtidos, e percebeu-se que o mesmo poderia servir como uma API de propósito geral, destinada a uma gama maior de programadores.

Assim, o SCV foi reformulado e encontra-se atualmente em sua terceira versão – mais informações sobre ele podem ser encontradas em [http://laca.inf.ufsm.br/scv/]. Esta versão estrutura-se de forma particularmente específica para programação orientada a objetos e proporciona uma maior gama de ferramentas ao programador. Faz-se necessário, no entanto, utilizar o SCV em um projeto de software com abrangência maior do que um trabalho de disciplina

acadêmica para validar definitivamente seu uso; e este trabalho se propõe, como um objetivo secundário, utilizar o SCV na construção de sua GUI para servir como caso de teste da API.

O SCV, na versão atual, é estruturado a partir de um núcleo (*kernel*), que é responsável por controlar todas as interações do usuário com a interface. Essa interface está modularizada, permitindo a criação de vários objetos, divididos em três grandes categorias que são passíveis de expansão e generalização: *containers*, *components* e *features*. O *container* é um painel onde podem ser adicionados objetos pelo programador; os *components* são os objetos que podem ser adicionados ao *container*; e *features* são recursos que não são visíveis ao usuário final do software, mas que podem ser usados pelo programador para facilitar o desenvolvimento de sua aplicação, abstraindo funções de baixo nível. Existem vários tipos de cada um desses objetos, que podem ser verificados na documentação disponível da API [http://laca.inf.ufsm.br/scv/] e, alguns, vistos em relação hierárquica no diagrama de classes simplificado, na Figura 6. Um exemplo de interface gerada com o SCV pode ser visto na Figura 7.

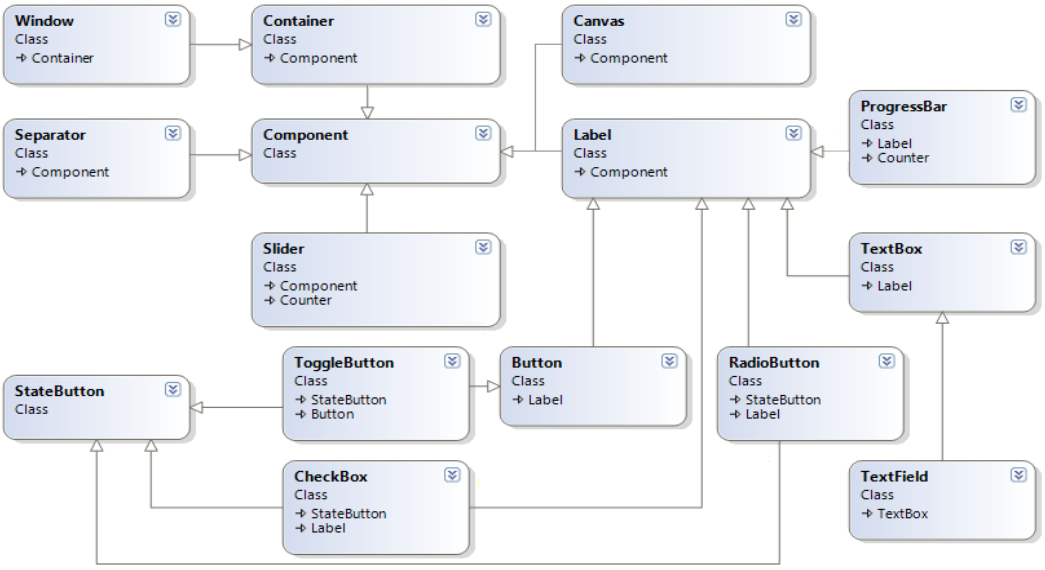

Figura 6 Diagrama de classes do SCV em versão atual.

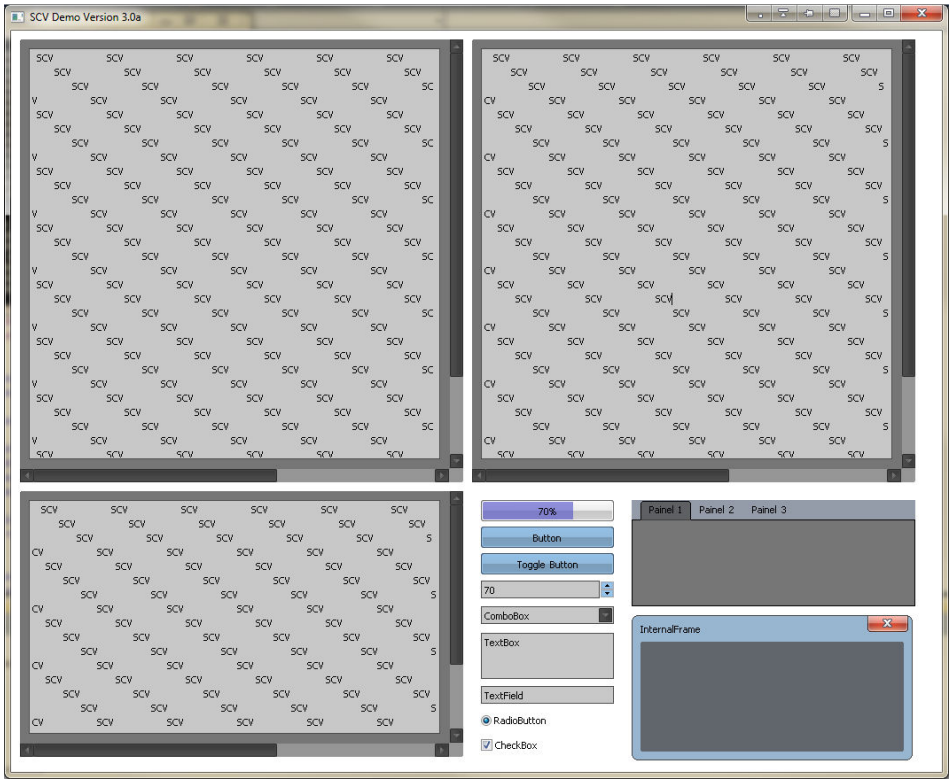

Figura 7 Exemplo de interface gerada com o SCV 3.0a.

Um componente de importância em especial é o *Canvas*: uma *viewport* do *OpenGL* através da qual o programador pode interagir facilmente para mostrar conteúdo através de comandos *OpenGL* abstraídos e simplificados, possibilitando o seu uso tanto em 2D como em 3D. Os *Canvas* tratam de maneira fácil as *Callbacks* de *mouse* e teclado e oferecem-nas ao usuário, além de já definir uma função de renderização – estes componentes são de especial interesse para o presente trabalho, conforme descrito na seção 4.2.3.1, por seu papel central na interação do software. Para melhor usufruto das *Callbacks* oferecidas pelo SCV, o tratamento do mouse e teclado são encapsulados, possibilitando ao programador interagir de forma natural e intuitiva com relação ao *input (mouse* e teclado) da aplicação.

Cada item de interface (seja ele um *canvas, button*, *textfield,slide bar, etc)* possui as próprias *callbacks*, desde que em sua definição o programador herde os métodos específicos, e o SCV se encarrega de definir qual componente tem o foco de mouse ou teclado em um dado momento e de realizar a chamada apropriada. Os métodos de *callback* disponibilizados para todos os componentes podem ser vistos na Figura 8. Assim, para criar um botão, por exemplo, o programador define uma classe descritiva derivada de *scv::Button* e declara as *callbacks* que deseja para

este botão, programando as ações que deseja serem tomadas em seu escopo. Durante a explanação do software, no capítulo 4, as *callbacks* aplicadas a cada componente utilizado serão descritas em maiores detalhes.

```
/*************************************************************************/
/* Mouse Callbacks                                                       */
/*************************************************************************/
virtual void onMouseClick(const scv::MouseEvent &evt);
virtual void onMouseHold(const scv::MouseEvent &evt);
virtual void onMouseOver(const scv::MouseEvent &evt);
virtual void onMouseUp(const scv::MouseEvent &evt);
virtual void onMouseWheel(const scv::MouseEvent &evt);

/*************************************************************************/
/* Keyboard Callbacks                                                    */
/*************************************************************************/
virtual void onKeyPressed(const scv::KeyEvent &evt);
virtual void onKeyUp(const scv::KeyEvent &evt);

/*************************************************************************/
/* Other Callbacks                                                       */
/*************************************************************************/
virtual void onResizing(void);
virtual void onDragging(void);
```

Figura 8 Detalhe de código mostrando as *callbacks* disponíveis para todo componente do SCV.

3 OBJETIVO DO SOFTWARE E PLANEJAMENTO

3.1 Motivação

Ao produzir software gráfico com o OpenGL, o programador depara-se com uma questão importante: a representação de modelos tridimensionais, também chamados modelos 3D. Esses modelos são representações matemáticas descritivas de superfíciese objetos em um espaço tridimensional, com amplo uso em variadas áreas da computação, mas que tem sua principal origem na computação gráfica. Os usos de modelos 3D são pertencentes a dois tipos: **exposição**, onde – geralmente – são transformados em imagens bidimensionais a partir de um processo conhecido como *rendering*; e **simulação**, onde os dados que caracterizam o modelo sofrem alguma operação de modificação (por exemplo, o modelo de um carro sofre modificações de força em pontos de estrutura que simulam um impacto) e, assim, configuram o resultado do processo (ou seja, seguindo o exemplo, é possível simular o resultado de um dado impacto em um carro).

Independente do uso, mas especialmente para a renderização, existem diferentes meios de obter modelos para uma aplicação tridimensional em OpenGL. O primeiro é definir os modelos com base nas primitivas gráficas que o OpenGL oferece. O segundo é a importação de modelos desenhados em softwares específicos - como o 3DStudioMax, por exemplo. Embora o segundo meio possibilite a descrição de objetos mais complexos com maior facilidade, ele também acarreta em algumas dificuldades que podem restringir ou impossibilitar seu uso.

Em primeiro lugar, as dependências geradas por processos de importação podem ser um fator negativo – como, por exemplo, a dependência a arquivos de dados auxiliares que encapsulam muito mais informação do que o necessário para a renderização simples, ou nem todas as informações desejadas. Além disso, em casos específicos (como trabalhos acadêmicos, por exemplo), a utilização de arquivos contendo os dados descritivos pode não ser possível, sendo absolutamente necessária a descrição dos objetos em puro código.

Em segundo lugar, softwares de modelagem costumam ser proprietários (com poucas exceções) e não possibilitam a exportação de código nativamente, requisitando plug-ins especiais que adicionam ainda mais complexidade. Isso torna necessário que o programador obtenha ou produza um interpretador para realizar a

tradução entre a linguagem de descrição própria do software de modelagem para código OpenGL. Além disso, como essa linguagem descreve os modelos de forma a otimizar o desempenho no próprio software de modelagem, o código obtido com esta tradução pode ser extremamente complexo e avesso ao modo como um programador o escreveria, dificultando sua livre edição. Isto deve-se ao fato de que os modelos, conforme construídos em softwares de modelagem, são compostos de forma a serem renderizados em *engines gráficas*, e o OpenGL "compreende" apenas vértices em coleções de linhas e polígonos.

Por fim, softwares de modelagem são complexos e requerem treinamento para a produção de modelos com coerência, em relação ao que é necessário para a renderização em OpenGL. Esses softwares são dedicados ao uso de modeladores e designers, e não programadores e estudantes de computação gráfica. Utilizam linguagem técnica e processos de manipulação que são intuitivos para designers, mas contra-intuitivos para os programadores sem noções avançadas de modelagem e processo artístico.

Todos estes problemas contribuem para que, especialmente nos casos em que o programador deseja modelos de complexidade baixa ou média, a importação de modelos seja preterida em favor da descrição direta em primitivas OpenGL. Esta, no entanto, apresenta as próprias desvantagens, especialmente relacionadas à composição de objetos que necessitam de muitos vértices e à dificuldade de prever (especialmente no caso de programadores mais inexperientes) o resultado gráfico de algumas operações do OpenGL.

A dificuldade de compor objetos com muitos vértices é resultado direto da necessidade de um número absurdamente alto de chamadas *glVertex*, talvez separadas em diversas chamadas de diferentes primitivas (chamadas *glBegin* e *glEnd*). Mesmo sem considerar dados de vetores normais para iluminação e coordenadas de textura, o esforço necessário para a descrição da posição de muitos vértices em código é um impedimento razoável à produção de modelos muito complexos desse modo. Além disso, conforme visto na seção 2.1.2, a ordem das chamadas é um fator importante no resultado obtido, e pode complicar ainda mais o processo de descrição de objetos com muitos vértices.

Quanto à dificuldade de prever-se os resultados de um dado código OpenGL na renderização final, ela é resultado da forma como interagem as múltiplas variáveis de estado do OpenGL. A cor de uma primitiva quando renderizada

depende, por exemplo, entre outras definições: da iluminação atual, composta de dezenas de variáveis de estado (número de luzes, posição de cada uma, valores de componente ambiente, especular e difusa de cada uma, vetores normais de cada vértice ou face de primitiva, modelo global de iluminação, *materials* aplicados aos objetos de cena e atributos de cada um deles); dos processos de *clipping* e *culling* aplicados e das opções adotadas para cada um deles; das opções de oclusão (estado do teste de oclusão, função de oclusão aplicada, dados de transparência); do modelo de interpolação de cor aplicado; etc. Conforme especificado na seção 2.1.1, Arquitetura da API OpenGL, todas estas definições possuem valores padrão, e o desconhecimento desses valores (inevitável devido a grande quantidade existente) acaba por confundir ainda mais o programador.

Assim, procurando oferecer um software que sanasse os problemas da descrição de objetos 3D em código OpenGL, projetou-se o software resultado do presente trabalho. Chamado primeiramente de *Drawtex*, em sua versão protótipo, foi renomeado *Drawtex SCV* na versão atual, a partir da qual passou a fazer uso da API SCV para a composição de interface gráfica. Os principais objetivos do software são discriminados seção 3.2, a seguir.

3.2 Objetivos

3.2.1 Objetivo geral

O objetivo principal do presente trabalho constitui-se na produção de uma ferramenta destinada ao programador C/C++, linguagens mais utilizadas em aplicativos que fazem uso de computação gráfica, utilizador da biblioteca gráfica OpenGL, especializada para a facilitação de descrição de objetos tridimensionais em código através da composição e visualização em interface interativa tridimensional, contando ainda comexportação de código parametrizável. Assim, o usuário pretendido é o programador OpenGL aprendiz, de modo que o software produzido sirva tanto como ferramenta de facilitação de desenho (com exportação de código OpenGL C/C++) quanto como ferramenta didática de Computação Gráfica, ao demonstrar visualmente os efeitos de cada variável ou atributo do OpenGL.

3.2.2 Objetivos específicos

Define-se como objetivos específicos de desenvolvimento:

- O foco na facilidade de uso, de acordo com o conhecimento e perfil esperado de um programador OpenGL;
- A construção modularizada de forma que o programa possa ser construído iterativamente e facilmente expandido;
- Implementar um modo de utilização que permita a definição de vértices no espaço em interface tridimensional, e GUI que permita a modificação dos atributos desses vértices;
- Permitir que o usuário visualize em tempo real os efeitos das mudanças de atributos e estados do OpenGL na renderização do modelo produzido;
- Permitir a exportação de código parametrizável de forma que o modelo produzido no software possa ser facilmente utilizado e editado em outras aplicações OpenGL construídas em C++;
- Estabelecer um controle interno do programa que aceite definições de preferências do usuário, carregadas a partir de arquivo de configurações;
- Estabelecer ainda a possibilidade de salvar e carregar arquivos de edição em sessões de uso posteriores;
- Permitir múltiplos contextos de edição (a edição de mais de um objeto) em uma mesma sessão de uso;
- Relatar as dificuldades e resultados obtidos com a implementação da GUI utilizando o SCV, para validar esta API;
- Disponibilizar o software gratuitamente quando da sua conclusão.

3.3 Planejamento e prototipação

De forma a verificar quais os pontos de dificuldade na implementação do software, foi produzido um protótipo em linguagem de programação C++ utilizando o OpenGL. Desejou-se explorar, principalmente, na construção desta versão inicial: o **modo de estruturação do programa**, que satisfizesse os requisitos de modularidade expostos nos objetivos; **o modo de interação do usuário** com o programa no espaço tridimensional; e, por fim, **o modo de exportação de código** a ser adotado.

Foi escolhido proceder a partir de planejamento com prototipação atrelada pelo fato de que é difícil planejar sem experiência prévia um software complexo, com

tantas possíveis abordagens de utilização, interação e saída de dados. Além disso, alguns dos conceitos que se sabia serem necessários para a aplicação (como edição tridimensional, encapsulamento do estado do OpenGL em estrutura de dados de aplicação, contextos de uso com carregamento de arquivos e preferências de usuário) são de implementação complexa e/ou trabalhosa; e a implementação final de um software costuma beneficiar-se da experiência adquirida na implementação de tais conceitos no protótipo [13].

O modo de estruturação inicial adotado para o protótipo pode ser visto no diagrama da Figura 9. Nele, pode-se averiguar que o programa é composto por um **MANAGER**, que contém gerenciadores específicos para cada parte do programa (gerenciador de iluminação, câmera[1] e entrada de dados). A única saída projetada inicialmente foi a renderização em tela, sem preocupação com o módulo de exportação de código - planejado a partir da análise do modo de estruturação, quando da conclusão da prototipação. Esse e outros resultados da análise são descritos na seção 3.3.1.

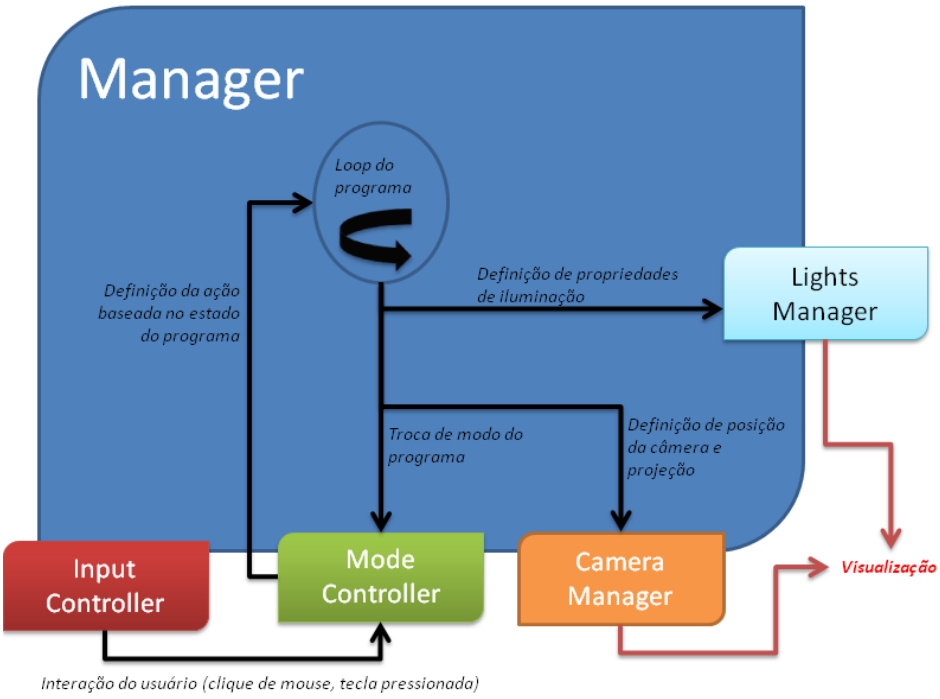

Figura 9 Diagrama simplificado do funcionamento do *DrawtexSCV*, protótipo.

[1] No que diz respeito ao *Drawtex* e ao *DrawtexSCV*, quando o texto referir-se a "câmera" como termo de implementação, ele refere-se a uma estrutura de dados que define todos os parâmetros de visualização referentes à posição do observador, direção da observação e parâmetros de projeção da cena.

O modo de interação do usuário com o programa foi definido como parte do gerenciador de entrada de dados (*input controller*). Como no protótipo não foi aplicada a API SCV, as *callbacks* utilizadas seguiram o modelo tradicional de aplicações OpenGL, fazendo uso das definições de funções de controle de mouse e teclado da GLUT. A Tabela 1 mostra um trecho da tabela dos comandos disponíveis de acordo com o modo corrente do programa - já em sua concepção original o *Drawtex* conta com múltiplas câmeras, mas alguns dos comandos descritos na tabela abaixo dizem respeito a funcionalidades implementadas após a análise, conforme descrito na seção 3.3.1.

No *Drawtex* original a classe *Manager* centraliza o controle do programa. Não foram criadas classes próprias para o gerenciador de câmera (*camera manager*) e de modo (*modecontroller*), sendo estes compostos por métodos da classe **MANAGER**, que segue o padrão de desenvolvimento de software *singleton*. A razão para a adoção deste*design pattern* deu-se pela necessidade de compartilhamento de recursos entre os módulos do programa, e de uma interface centralizadora de controle de acesso a esses recursos [14] – papel desempenhado pelo *singleton*– evitando a utilização de variáveis globais.

Tabela 1Tabela parcial de comandos de usuário do Drawtex protótipo.

Entrada	Modo de edição	Modo de visualização
Barra de espaço	alterna câmera ativa	alterna câmera ativa
F1	liga/desliga guias visuais	liga/desliga guias visuais
***Numpad*1,2,3**	ativa câmera correspondente	ativa câmera correspondente
U,I	modifica ângulo de abertura da câmera atual	modifica ângulo de abertura da câmera atual
O,P	modifica aspecto de projeção da câmera atual	modifica aspecto de projeção da câmera atual
J,K	modifica campo-de-visão da câmera atual	modifica campo-de-visão da câmera atual
Q	deleta todos os vértices desenhados	deleta todos os vértices desenhados
Z	troca de modo	troca de modo
Setas direcionais		move o vértice no eixo X-Z
Clique meio (*mouse*)	alterna câmera ativa	alterna câmera ativa
Arraste esquerdo	*pan* da camera	*pan* da camera
Arraste direito	rotação da camera	rotação da camera
Clique esquerdo	toma posição inicial do vértice para adição no ponto de clique	seleciona vértice no ponto de clique
Arraste esquerdo	modifica posição do vértice tomado para adição, no eixo Y	

3.3.1 Análise do protótipo

Percebeu-se que seria muito trabalhoso produzir código C/C++ fazendo uso das estruturas de dados internas do programa, pelo fato de que essas estruturas, neste protótipo, não seguem o padrão do OpenGL; ou seja, as variáveis de estado da API são obtidas pelo *mode controller* durante o passo de "*Definição da ação baseada no estado do programa*", segundo a Figura 9, mas não são guardadas em memória de aplicação. Assim, definiu-se para o programa final que as estruturas de dados internas deveriam refletir ao máximo o código a ser exportado, de forma a facilitar a tradução. Mais detalhes sobre esta paridade entre os dados internos do DrawtexSCV e o código exportado são dados nas seções 4.2.2.3 e 4.2.3.2.

Além das conclusões referentes à exportação de código, obtiveram-se alguns resultados em relação ao modo de estruturação. Percebeu-se que, para permitir definições de preferência do usuário, seria necessária a inclusão de outro módulo específico para esta tarefa. Além disso, percebeu-se que os controladores de câmera e iluminação poderiam ser combinados de forma a centralizar todo o controle de visualização em uma única classe. Foi comparando os resultados obtidos neste quesito com as ferramentas proporcionadas pelo SCV que a estrutura aplicada no programa final foi desenvolvida, conforme descrito na seção 4.1.

Por fim, quanto ao modo de interação do usuário no espaço tridimensional, foi possível definir uma forma de criação/edição de vértices e navegação a partir do mouse. Inicialmente, foi testada a abordagem de interface por linha de comando, e depois, por componente de interface gráfica (entrada de valores em caixas de texto). No entanto, espelhando-se no modo de funcionamento de softwares de modelagem, que apesar de foco de uso diferente[2] apresentam navegação e edição em espaço 3D intuitiva, percebeu-se que uma usabilidade mais apropriada envolveria o mouse e o uso da técnica de *picking*, aliado a uma discretização do espaço em um modelo de grade – isto é, demonstração gráfica da separação do espaço de forma que cada célula da grade corresponda, quando da exportação de código, a uma unidade de coordenada do OpenGL[3]. Assim, cada vértice definido pelo usuário pode ser

---

[2] Softwares de modelagem tridimensional conforme os citados anteriormente preocupam-se com a descrição de primitivas (planos, linhas, curvas, superfícies, etc.) em composição de modelos; e não com a posição ou ordem de aplicação de vértices individuais no espaço.

[3] Unidades de coordenada são definidas de acordo com a geometria do programa, no OpenGL, e são definidas arbitrariamente. Requer-se do programar que ele responsabilize-se por manter a projeção e

representado por um cubo de lado unitário (i.e, do mesmo tamanho que uma célula da grade) – uma abstração necessária para a visualização, embora o vértice, no OpenGL seja uma abstração matemática de um ponto sem dimensões, conforme estabelecido na seção 2.1.2.

Outra contribuição do protótipo para a implementação final, também baseada na interface dos softwares de modelagem, foi a utilização de guias de auxílio. Interagir com objetos em um espaço tridimensional pode ser confuso sem pontos de referência nas três dimensões e, portanto, há várias guias visuais que auxiliam o usuário a perceber a posição relativa entre os elementos no espaço. As guias desenvolvidas para o protótipo incluem: guias de câmera; guias de eixo espacial; guias de grade e guias de desenho. A Figura 10 mostra as guias em utilização do *Drawtex* original. Destaca-se:

1. O grid de desenho, dividindo o espaço de modo discreto, conforme especificado acima, e os eixos de coordenadas;
2. Guias de visualização, ajudando a definir os campos-de-visão das câmeras inativas e posição dos vértices no eixo *Y*;
3. Guias de desenho, ajudando a definir a posição do vértice a ser desenhado nos eixos *X,Y* e *Z*.

Ainda espelhando-se nos softwares de modelagem, foi possível constatar que a interface de desenho beneficia-se muito com a presença de *viewports* com projeção ortogonal de visão superior e lateral da cena de edição. Tanto a técnica de *picking* como a visualização em *viewports* auxiliares são descritas na seção 4.2.3.1. Uma cena de visualização produzida pelo programa pode ser vista na Figura 11.

---

as transformações adequadas às coordenadas que utiliza; e também que mantenha os valores de diferentes coordenadas condizentes com seus propósitos [opengl.org – OpenGL FAQ].

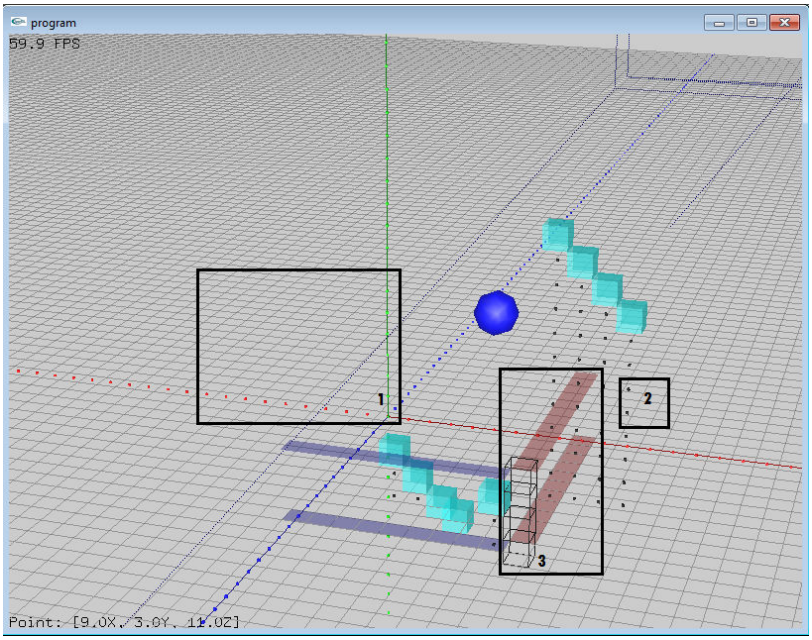

Figura 10 Visualização produzida pelo protótipo, com guias destacadas.

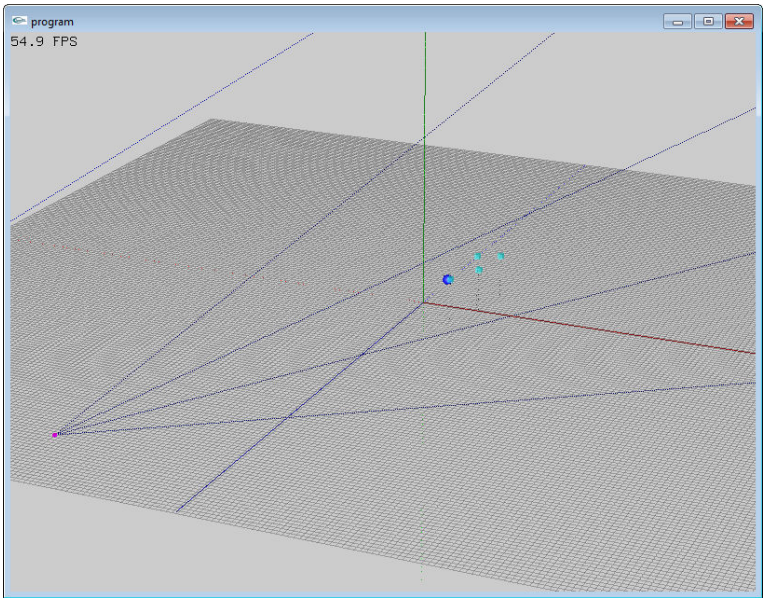

Figura 11 Visualização geral do protótipo.

4 DRAWTEX SCV

4.1 Estrutura do programa

A partir dos resultados obtidos com o protótipo, foi planejado o modelo de estruturação do *DrawtexSCV*. Uma importante decisão neste aspecto foi a adoção do SCV, em vista do objetivo específico de validação da mesma. O interesse na API vem da ligação do acadêmico realizador do presente trabalho com a mesma, tendo participado do projeto de sua primeira versão e publicado, junto aos participantes das edições posteriores, artigos sobre esses projetos. Na estruturação do programa, além da validação do SCV, levou-se em conta como principal objetivo a modularização do software, para possibilitar sua construção iterativa e facilitar sua expansão. Esta modularização baseou-se na separação dos processos que são o foco do usuário do *DrawtexSCV*em três classes lógicas:

- A **edição** de objetos, focada na criação e manipulação de vértices;
- A **visualização** do resultado da edição de acordo com parâmetros do usuário;
- e a **exportação** do código OpenGL, também parametrizável.

Nota-se que cada uma delas diz respeito a um dos objetivos específicos do software, conforme estabelecido na seção 3.2.2.

Assim, de acordo com a experiência adquirida na construção do protótipo, decidiu-se estabelecer módulos controlados por uma classe central, definida no modelo *singleton* – buscando, da mesma forma que na implementação anterior, evitar a replicação de dados e controlar o acesso a recursos compartilhados. Os módulos, ou cenas, foram classificados como `Scenes` e a classe gerenciadora como `Program`. Nenhuma `Scene` interage com a outra, exceto através do *singleton*, de forma que as comunicações entre elas são controladas e os recursos compartilhados por todas ou são parte do `Program` ou acessíveis apenas através dele. A Figura 12 mostra essa estrutura básica. Observa-se que o `Program` possui como atributo privado uma coleção de cenas, organizada em um vetor. O atributo privado `activeScene`, por sua vez, determina qual a cena ativa em um dado momento. Detalhes sobre a implementação das cenas são dados na seção 4.2.3.

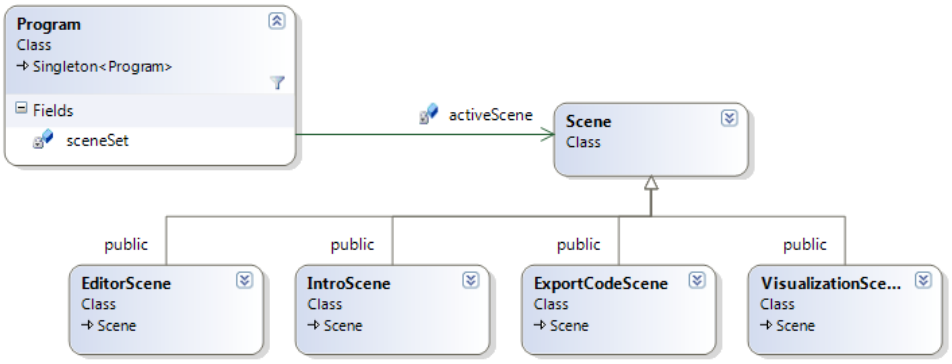

Figura 12 Estrutura básica do *Drawtex SCV*, em diagrama simplificado, ilustrando a relação entre a classe *singletonProgram* e as *Scenes.*

Aos recursos compartilhados acessíveis apenas através do **PROGRAM** dá-se o nome, no *DrawtexSCV*, de *contextos de usuário,* representados pela classe **USERCONTEXT** e derivadas. Esta é uma diferença crucial entre o modo de estruturação do protótipo e da versão atual: nesta última, os recursos compartilhados configuram-se como conjuntos de dados abstraídos em classes, e não apenas como os vértices definidos pelo usuário e seus atributos, como no protótipo. Os contextos de usuários definidos inicialmente encapsulam os dados necessários para permitir as ações referentes aos processos de edição, visualização e exportação definidos anteriormente. Foram chamados de*contexto de edição*, *opções de visualização* e *opções de* exportação, respectivamente. A Figura 13 mostra a relação entre o programa, as cenas e os contextos de usuário.

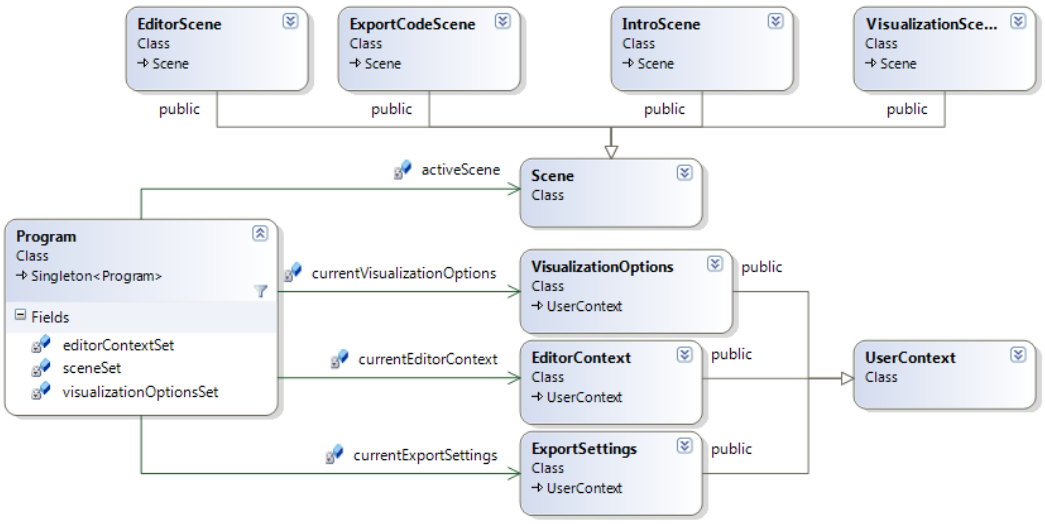

Figura 13 Diagrama simplificado mostrando a relação entre **PROGRAM**, as **SCENES** e os **USERCONTEXT**.

O **PROGRAM** possui uma coleção de **EDITORCONTEXT** (a classe que encapsula o *contexto de edição*), o **editorContextSet**, de forma a permitir que o usuário possa alternar entre contextos na mesma execução do *DrawtexSCV* e desenhar/visualizar composições diferentes sem ser necessário iniciar uma nova sessão de uso ou carregar os dados de um arquivo. Além disso, o **PROGRAM** possui – como atributo privado, para controlar sua modificação – uma referência ao contexto atual, o **currentEditorContext**, a partir do qual toda comunicação dos outros módulos com este conjunto de dados é mediada. Detalhes sobre os atributos e métodos dos contextos de edição são dados na seção 4.2.2.2. O modo como as diferentes cenas interagem com o contexto de edição ativo, através da interface oferecida pelo *singleton,* é descrito ao longo do texto do capítulo 4.

De forma correlata, outra coleção contida no **PROGRAM** é o **visualizationOptionsSet**, e outra referência privada é o **currentVisualizationOptions***;* ambos relacionados aos **VISUALIZATIONOPTIONS**, que determinam principalmente os parâmetros para a cena de visualização. As opções de visualização são descritas na seção 4.2.2.1.

Por fim, ao contrário dos casos anteriores, o **PROGRAM** não possui uma coleção de **EXPORTSETTINGS**, mas apenas uma referência direta a uma instância única, o **currentExportSettings**. Esta referência é privada, pelas mesmas razões expostas acima, nos casos das referências aos outros contextos atuais de usuário. A razão para não haver uma coleção de parâmetros de exportaçãoé que foi considerado desnecessário alternar entre diferentes conjuntos de parâmetros rapidamente, ao contrário dos casos dos contextos de edição e opções de visualização. O *DrawtexSCV* permite, no entanto, que os parâmetros de exportação sejam salvos como parte das preferências de usuário.

Ainda sobre a estrutura do programa, é necessário esclarecer o modo de entrada de dados e o fluxo dos processos a partir da interação do usuário com a interface gráfica. A aplicação do SCV impacta no modo como o programa lida com as entradas de usuário e demais *callbacks*, conforme mencionado na seção 2.2. Especificamente, definiu-se que os componentes fornecidos pelo SCV seriam todos atrelados às **SCENES**. Assim, toda área interativa do *DrawtexSCV* é um componente de interface ligado à uma cena, com exceção dos *popups*, conforme visto adiante.

O fluxo de tratamento da entrada de dados é exposto na Figura 14. Nesta, as setas representam o fluxo de dados através dos componentes do sistema. É o SCV o encarregado de tratar os *eventos de mouse e teclado* **(1)**; e de *gerenciar qual o componente ativo no momento* **(2)**; o componente então ou *atua na própria cena* **(3a)** que então *atualiza o programa* **(3b),** ou *atualiza o programa diretamente* **(4)**. Por fim, o *programa atualiza os contextos de usuário necessários* **(5)**. É preciso notar que as atualizações de programa já definem quais os contextos de usuário a serem modificados; estas chamadas já são definidas no componente ativo (ou seja, é a ação do componente ativo que determina qual método de interfaceamento será chamado no *singleton*).

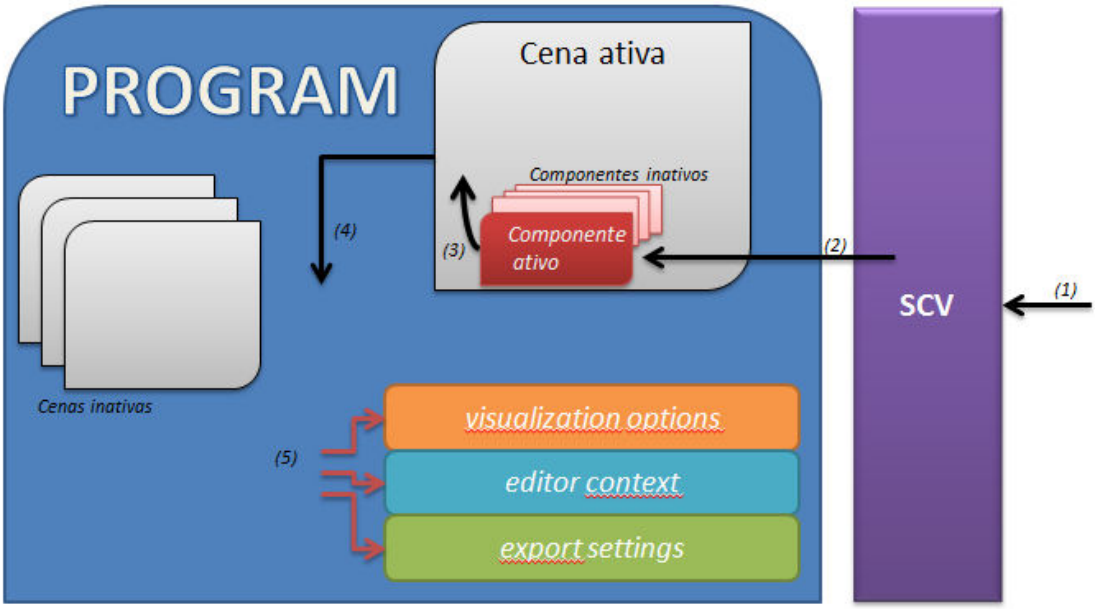

Figura 14 Fluxo de dados a partir da entrada do usuário no *Drawtex SCV*

Além das cenas, os componentes de interface oferecidos pelo SCV também podem estar inclusos em *popups*. Cada *popup* é uma janela flutuante, ela mesma um componente do SCV, que pode ficar ativa durante a transição entre duas cenas e contém os próprios componentes, codificados na classe **PopUp**. Neste quesito, um **PopUp** é similar a uma cena; e, desta forma, a explicação sobre o fluxo de dados a partir da entrada do usuário estende-se naturalmente a ele. Um exemplo de *popup* utilizado é o **LightOptionsPopUp**, derivado da classe **PopUp**, que contém componentes que controlam parâmetros de visualização ligados à iluminação. Um PopUp da interface pode ser visto em destaque na Figura 15.

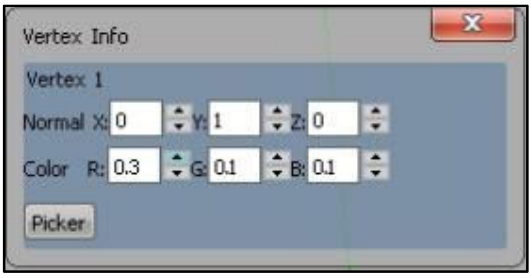

Figura 15 Um exemplo de PopUp existente no programa.

4.2 Implementaçao

4.2.1 Program

É na classe *Program* que são implementados os métodos de controle e interfaceamento entre os módulos (cenas) do *DrawtexSCV*. Conforme especificado anteriormente, esta classe segue o padrão de desenvolvimento de *singleton,* isto é, uma única instância da classe pode existir e o acesso a ela é controlado pela própria classe. A implementação de *singleton* adotada é a oferecida pelo próprio SCV, que pode ser vista na documentação da API. Os métodos do Program estão divididos em sete grupos. São eles: relacionados ao SCV; de fluxo do programa; utilitários; de interface dos **UserContexts** atuais; e de controle dos *popups*. As seções a seguir tratam das classificações e de alguns casos específicos dentre cada uma delas.

4.2.1.1 Relacionados ao SCV

Estes são, basicamente, métodos de acesso ao **Kernel** do SCV e inserção de componentes no contexto da API. São listados na Figura 16.

```
//SCV-related
scv::Kernel* getKernel(){return scv::Kernel::getInstance();}
void runKernel(){ scv::Kernel::getInstance()->run();}
void addComponentToKernel(scv::ComponentInterface *component){scv::Kernel::getInstance()->addComponent(component);}
```

Figura 16 Trecho de código mostrando os principais métodos relacionados ao SCV na classe Program.

O método **getKernel** é acessível externamente (público), e é utilizado quando alguma ação de componente necessita modificar algum parâmetro ou o contexto do SCV diretamente. Não é utilizado no programa, exceto para modo de teste e *debugging* (isto é, o programa final não faz uso dele em momento algum). Ele retorna a instância (também em *singleton,* por definição dos SCV) do **Kernel** de controle do SCV.

Já o método **runKernel** é chamado na função **main** do programa, logo após a preparação do programa com o carregamento das preferências de usuário e inicialização das cenas e estruturas de dados do **PROGRAM**. Equivale, em uma aplicação com o SCV, à chamada **glutMainLoop**, a partir da qual o laço do OpenGL entra em execução e as callbacks são ativadas.

4.2.1.2 Fluxo do programa

Os métodos de fluxo do programa dividem-se em de controle de cena e de construção e inicialização. Os últimos são aqueles chamados apenas quando da inicialização do *DrawtexSCV*, que carregam as opções usuário e criam as estruturas de dados necessárias. Os métodos de controle de cena são os responsáveis por ativar e desativas as cenas e todos os seus componentes. Estes também estão listados na Figura 17.

```
//Flow
	//starting
	void startMessage(void);
	void loadPreferences(void);
	void startScenes();
	//Scene control
	void goToVisualizationScene();
	void deactivateScene();
	void activateScene(std::string theTag);
```

Figura 17 Trecho de código contendo a declaração dos métodos de fluxo da classe Program.

O método **startMessage** imprime a mensagem de inicialização no console após a leitura das preferências do usuário, apontando algum eventual erro – e, nesse caso, terminando a sessão do programa. O método **loadPreferences** é o responsável pela leitura do arquivo de inicialização, em um formato pré-definido, e parametrizar a criação das cenas e demais estruturas de dados. Em caso de erro, comunica o método **startMessage** em que linha do arquivo e qual o tipo de erro encontrado.

É o método **startScenes** que organiza as chamadas de criação de todas as demais estruturas de demais estruturas de dados básicas do software; três delas referem-se a métodos muito similares, muito similares, também classificados nesta mesma categoria: **createLightOptionsPopUp**, **createLightOptionsPopUp**, **createAppliedPointexOptionsPopUp** e **createPointexInfoPopUp**. O método **createLightOptionsPopUp** é descrito na

Figura 18.

```
/*********************************Pop up management*****************************/
void Program::createLightOptionsPopUp(){
   popUpSet.push_back(new LightOptions_popup(5*DEFAULT_GRID_CELL_WIDTH+
                                             4*DEFAULT_HORIZONTAL_GRID_BORDER,
                                             12*DEFAULT_GRID_CELL_HEIGHT+
                                             11*DEFAULT_VERTICAL_GRID_BORDER));
   popUpSet.back()->setVisible(false);
   popUpSet.back()->setRelativePosition(getGridPoint(3,12));
   scv::Kernel::getInstance()->addWindow(popUpSet.back());
}
```

Figura 18 Trecho de código do método responsável pela criação de um dos *popups* do programa.

Quando da chamada de um método de criação de *popup*, conforme mostra a figura, é inserido um novo **PopUp**, já instanciado em uma classe derivada adequada, no conjunto de **PopUps** do **Program** (**popUpSet**) – no caso, um **LightOptions_popup**.

Os parâmetros do construtor do método em questão são indicativos do modo de construção de interface semi-dinâmica adotado no software, baseado em valores absolutos definidos em código que definem um *grid* de células nos quais todos os componentes de GUI se encaixam, quando inicializados. Este modo também se relaciona com o método estático do **Program getGridPoint**, utilizado na criação de componentes para o cálculo de posição inicial baseado neste *grid* e explicado na seção 4.2.1.3, abaixo.

Por fim, todos os métodos de controle de cena lidam apenas com a referência à cena atual, **activeScene**, e operam uniformemente sobre as cenas que precisam ser ativadas/desativas – ativando/desativando todos os seus componentes. Foi na implementação destes que se percebeu uma das deficiências do SCV atual: a incapacidade de deletar componentes de interface (e liberar memória de aplicação). Assim, os métodos de desativação de cenas e **PopUps** limitam-se a esconder todos eles com chamadas **setVisible,** quando seria possível deletar aqueles que não são mais necessários. Foi implementado, no entanto, um modo de controle para que componentes desnecessários fossem marcados como tal e modificados para substituir novos componentes do mesmo tipo, quando necessário.

4.2.1.3 Utilitários

Além destes, o **Program** também contém alguns métodos de utilidade, em geral estáticos por serem recorrentemente necessários em vários módulos e não acessarem nenhuma estrutura de dado de instância. Em geral são métodos que lidam com a composição da interface ou com processos que precisam ser padronizados ao longo da execução (como, por exemplo, o arredondamento de valores de ponto flutuante, descrito a seguir).

Um destes métodos permite a troca do título, na janela, da aplicação (**setWindowTitle**). Outros permitem a escrita bidimensional no espaço 3D (**write2D**, **write2DColor**). Outros ainda realizam diferentes operações numéricas: de arredondamento (**roundAwayFromZero**, **roundTowardZero**), necessários de modo constante durante o programa; ou de comparação com limiar (**isSimilarXZ**, **isSimilarY**), usados especialmente na comparação de coordenadas em espaço tridimensional.

Por fim, métodos auxiliares de construção de interface, **getGridPoint** e **getGridPointf**, conforme citado na seção anterior. Estes métodos dizem respeito à separação do espaço da tela em um *grid* bidimensional de células, nas quais os componentes de interface se encaixam quando de sua criação. Assim, modificando-se os valores que alteram as dimensões da tela ou das células do *grid* mantém-se constante a composição da interface. As dimensões padrão definem o *grid* como tendo nove células de largura por trinta e duas de altura. Assim, um componente criado com a posição definida por **getGridPoint**(0,0) – os parâmetros sendo a linha e coluna do *grid* – sempre será gerado no topo esquerdo da tela, mesmo que as dimensões padrão sejam modificadas.

4.2.1.4 Interface de UserContexts

Os métodos de interfaceamento do **EditorContext**, **VisualizationOptions** e **ExportSettings** são encaminhamentos diretos, geralmente pareados com algum método de um destes contextos de usuário - ou seja, realizam algum tipo de controle ou teste e depois chamam a função correta no contexto de usuário apropriado. Deve-se considerar, para fins de estruturação do software, que todo método público de um UserContext é chamado apenas através destes métodos, e que esta chamada sempre existe.

4.2.1.5 Interface de **PopUps**

O programa ainda fornece uma interface de gerenciamento e controle dos **PopUps**, que são, conforme visto anteriormente na seção 4.1, elementos de GUI que flutuam sobre os demais. **Popups** são janelas que contém outros componentes e que se relacionam com o **Program** de modo a afetar os contextos de usuário, sendo mostrados por sobre as cenas e de forma constante nas trocas de cena ativa, salvo

exceções (a cena **ExportCode**, por exemplo, não permite *popups,* por razões expressas na seção referente a implementação desta cena, 4.2.3.2). Comportam-se de modo similar aos métodos de controle de cena.

4.2.2 UserContexts

4.2.2.1 VisualizationOptions

O objetivo principal do contexto de usuário **VisualizationOptions** é encapsular os parâmetros de visualização do DrawtexSCV. Também é a classe mais ligada ao objetivo de fornecer um modo de compreensão sobre como as variáveis de estado do OpenGL se relacionam e o produto visual de sua composição. Assim, os parâmetros contidos nesta classe devem satisfazer a dois objetivos do trabalho: primeiro, permitir que o usuário aprendiz de OpenGL verifique visualmente qual é o resultado da aplicação ou parametrização de cada um dos conceitos ligados à visualização do OpenGL (luzes, materiais, funções de *culling*). Além disso, prevendo a necessidade de exportação de código e levando-se em conta a experiência da implementação do protótipo, estes parâmetros devem ser guardados de forma coerente com os dados que representam no contexto do OpenGL.

Uma das possibilidades oferecidas pela inclusão de múltiplos **VisualizationOptions**, com o ativo alternante, no **Program**, é a capacidade do usuário definir, em um deles, os parâmetros de visualização de sua própria aplicação – ou seja, da aplicação na qual planeja utilizar primitivas construídas no *DrawtexSCV* – e em outro, modificar estes parâmetros para critério de comparação. Assim não apenas é possibilitado a ele que veja o resultado da importação de uma primitiva produzida no *DrawtexSCV* em sua aplicação, como também lhe é fornecida uma interface na qual pode verificar visualmente se os parâmetros iluminação, material, etc. de sua aplicação poderiam ser melhorados para os resultados que espera. Alguns dos atributos contidos em um VisualizationOptions podem ser vistos na Figura 19. Outros atributos importantes definidos nesta classe são: o **`clearColor`**, a cor com a qual o os*ColorBuffers* são preenchidos quando da chamada *glClear* passando-se como parâmetro *GL_COLOR_BUFFER_BIT* (algo comumente realizado no começo de cada método ou função de renderização); **`shadeModel`**, que define o modo de interpolação de cor aplicado entre os vértices; e

currentPrimitive, que define qual a primitiva a ser passada como parâmetro na chamada *glBegin* nas visualizações do resultado dos vértices editados.

```
/***************** LIGHTMODEL & MATERIAL ************************************/
/*GL_LIGHTING (GL_FALSE)      */                          bool enableLighting;
/*GL_COLOR_MATERIAL (GL_FALSE)*/                          bool enableMaterial;
/*GL_COLOR_MATERIA_PARAMETER (GL_AMBIENT_AND_DIFFUSE)*/   GLint materialParam;
/*GL_COLOR_MATERIAL_FACE (GL_FRONT_AND_BACK)*/            GLint materialFace;
/*GL_AMBIENT (xyz0.2,1.0)*/                               GLfloat materialAmbient[4];
/*GL_DIFFUSE (xyz0.8,1.0)*/                               GLfloat materialDiffuse[4];
/*GL_SPECULAR (xyz0.0,1.0)*/                              GLfloat materialSpecular[4];
/*GL_EMISSION (xyz0.0,1.0)*/                              GLfloat materialEmission[4];
/*GL_SHININESS (0.0)*/                                    GLfloat materialShininess;
/*GL_LIGHT_MODEL_AMBIENT (xyz0.2,1.0)*/                   GLfloat lightModelAmbient[4];
/*GL_LIGHT_MODEL_LOCAL_VIEWER (GL_FALSE)*/                bool lightModelLocalViewer;
/*GL_LIGHT_MODEL_TWO_SIDE (FALSE)*/                       bool lightModelTwoSide;
/*GL_LIGHT_MODEL_COLOR_CONTROL (GL_SINGLE_COLOR)*/        GLint lightModelColorControl;
```

Figura 19 Parâmetros de um VisualizationOptions referentes à iluminação e *materials*.

Os métodos de um **VisualizationOptions** são métodos de acesso e definição de seus atributos; e métodos de aplicação de seus próprios atributos para desenho. Na primeira classificação enquadram-se todos os métodos *set* e *get* para todos os atributos que classe encapsula. Os métodos de *set* são ativados através de métodos equivalentes na interface de chamadas do Program e atualizam os valores dos atributos (geralmente, em consequência do processamento de uma entrada de usuário, conforme visto na seção 4.1.

Já os métodos de aplicação de atributos são aqueles que atualizam o estado do OpenGL de acordo com os valores dos atributos da classe. Exemplos são os métodos **applyClearColorOptions** (que realiza a chamada *glClearColor* passando os valores de seu atributo **clearColor**); **aplyEnableDepthTestOption** (que realiza um teste e, de acordo com o valor booleano do atributo **enableDepthTest**, chama *glEnable*ou *glDisable* passando como parâmetro *GL_DEPTH_TEST);* e applyLightingOptions, que realiza um série de testes e definições de propriedades de iluminação baseada nos atributos mostrados na Figura 19.

4.2.2.2 Editor Context

O EditorContext atual mantém uma relação de quais os vértices criados pelo usuário, o **pointexSet**, além de uma relação da ordem na qual esses vértices são aplicados para a formação da primitiva, **appliedOrderSet**. Cabe lembrar que os vértices podem ser repetidos em aplicação, e, portanto, o **appliedOrderSet** não é uma sequência de ponteiros para os vértices guardados em **pointexSet**; mas sim uma sequência dos índices deles na ordem em que devem ser aplicados. A interface

que lida diretamente com esta sequência é a do PopUp “Vertex Order”, que pode ser visto na Figura 20. Também contém uma variável de controle que dita se todo vértice desenhado é automaticamente aplicado em **`appliedOrderSet`**. Ao contrário do **VisualizationOptions**, que possui métodos de diferentes tipos, o EditorContext possui apenas métodos de acesso e definição de seus atributos, pareados na implementação do **Program** e acessados só através destes pares.

Figura 20 PopUp de Vertex Order, a partir do qual o usuário insere, remove e edita a ordem de aplicação dos vértices existentes.

Cabe destacar que, apesar dos tipos de dados guardados pela aplicação serem, ao máximo possível, espelhados nos valores guardados pelo OpenGL, o vértice desenhado pelo usuário é uma abstração maior que apenas coordenadas do espaço. Assim, é representado por uma classe completa, contendo dados de posição, cor e normais para um dado vértice. Essa classe denomina-se **PointEx**, no contexto do SCV, mas sua importância é estritamente interna e não é transparente ao usuário.

4.2.2.3 Export Settings

De forma similar ao EditorContext, todos os métodos do ExportSettings são espelhados por métodos de interface em Program; e todos lidam com o acesso e modificação de atributos da classe. Os atributos, porém, neste caso, são as parametrizações para a exportação de código. Algumas das funcionalidades de exportação de código pretendidas pelo DrawtexSCV incrementam o número de parâmetros desta classe, mas alguns dos principais são descritos a seguir. São eles: os fatores de modificação de coordenadas, opção de normalização de vetores normais, e tipos de chamada *gl*.

Os fatores de modificação de coordenadas são expressos pelos atributos **`xFactor`**, **`yFactor`** e **`zFactor`**. Conforme explicado anteriormente na seção 3.3.1, o espaço discretizado permite que o vértice criado pelo usuário seja representado por um cubo de lado unitário. Assim, como dentre as opções de usabilidade encontra-se a proibição de definição de múltiplos vértices no mesmo ponto (pois a replicação de aplicações sana todas as necessidades desta replicação de vértices); foi necessária a inclusão destes fatores para que o código gerado não ficasse limitado a valores inteiros de coordenadas para os vértices. O efeito dos fatores no código e a multiplicação deles em todas as coordenadas do tipo que representam. Assim, com os fatores xFactor e zFactor mantidos em 1 (a opção padrão) e yFactor setado para 0.3, os valores de coordenada *X* e *Z* dos vértices no código exportado serão sempre múltiplos de 1; e os valores de coordenada de Y serão multiplicados por 0.3 e assim, os valores 0.3 e 1.2 podem ser alcançados, por exemplo, desenhando-se na primeira e quarta células neste eixo, respectivamente. Além disso, a utilização de fatores multiplicativos permite que o usuário desenhe com os vértices próximos, na interface de edição, mas que geram primitivas muito grandes na exportação de código.

A opção de normalização dos vetores normais foi incluída pelo fato de que esta é uma operação obrigatória, no OpenGL, para garantir a simetria e qualidade da iluminação virtual, e que pode gerar problemas quando desapercebida. É consenso que é obrigação do programador fornecer os dados dos vetores normais já normalizados ao OpenGL, mas é fornecida uma maneira – muito custosa computacionalmente, no entanto – de estabelecer que todo vetor normal seja normalizado antes da aplicação (setar a variável de estado do OpenGL relevante

com uma chamada *glBegin* passando como parâmetro a constante *GL_NORMALIZE*). Como o DrawtexSCV se propõe a exportar código que inclui informação de normais, considerou-se necessário realizar este processo de normalização *off-line*, isto é, na geração de código. Assim, esta normalização é oferecida como opção ligada por padrão.

Os tipos de chamada *gl* estabelecidos por parâmetros de **ExportSettings** incluem a primitiva a ser passada como parâmetro na chamada *glBegin*, a forma da chamada *glVertex*, que determina o tipo de parâmetro que recebe, e a forma da chamada *glNormal*. Estas chamadas podem receber, por exemplo, dois parâmetros inteiros (*glVertex2i*) ou três parâmetros *double* (*glVertex3d*), ou mesmo quatro parâmetros de ponto-flutuante (*glVertex4f*). O número de parâmetros diz respeito às coordenadas *x*, *y*, *z* e *w;* sendo a última utilizada como coordenada homogênea e de valor, em caso de omissão, igual a um. A coordenada *z,* por sua vez, quando omitida, recebe o valor zero.

4.2.3 Cenas

Criadas na inicialização, as cenas são os módulos lógicos do programa, nos quais os componentes de interface são ligados. São as cenas que interagem com a classe **Program**, e cujos componentes (ao lado daqueles dos **PopUps**) determinam as modificações dos atributos dos **UserContexts** descritos nas seções precedentes.

As cenas fazem uso dos métodos de Program que compõe a interface dos métodos dos contextos de usuário tanto para modificar estes atributos quanto para atualizar seus próprios estados internos, renderizações e componentes baseados neles. Além disso, os componentes das cenas comunicam-se com o **Program** para modificar dados de **PopUps** (ativá-los e desativá-los, especialmente).

Em vista do papel especial que desempenham, é importante descrever os componentes **Canvas** utilizados no *DrawtexSCV*. Derivados da classe homônima do SCV, os **VisualizationCanvas**, **EditorSceneCanvas** e **IntroCanvas** possuem as próprias callbacks e papéis peculiares no software. O IntroCanvas, por exemplo, é presente apenas em na cena de introdução, a **IntroScene**, na qual recebe uma textura de fundo de tela para efeito estético. O **EditorSceneCanvas** só é instanciado como atributo da **EditorScene**, assim como os **PortletViewCanvas**, derivados deste último. O **VisualizationCanvas**, no entanto, é instanciado na **VisualizationScene**, onde configura-

se como o componente que ocupa toda a tela, também é utilizado de forma similar aos **PortletViewCanvas** na cena de edição.

Todas as cenas possuem um rol de componentes (**componentSet**), um rol de imagens (**imageSet**) e um conjunto de **Canvas** (**canvasSet**). Os métodos da classe **Scene** são os métodos de ativação e desativação (**activate**, **deactivate**) e controle de acesso (**lock**, **unlock**).

4.2.3.1 EditorScene

A cena de edição é onde o usuário cria e modifica os vértices (**PointExs**) que são utilizados para a composição de primitivas e exportação. A interação do usuário segue todos os conceitos anteriormente abordados, especialmente na seção de análise do protótipo, 3.3.1. Além da separação do espaço de desenho em grade e das guias de desenho, modificadas para esta versão, figura uma das características mais importantes do *DrawtexSCV*: a interação no espaço 3D, implementada através da técnica de *Picking*. Uma visualização geral da cena de edição padrão é mostrada na Figura 21.

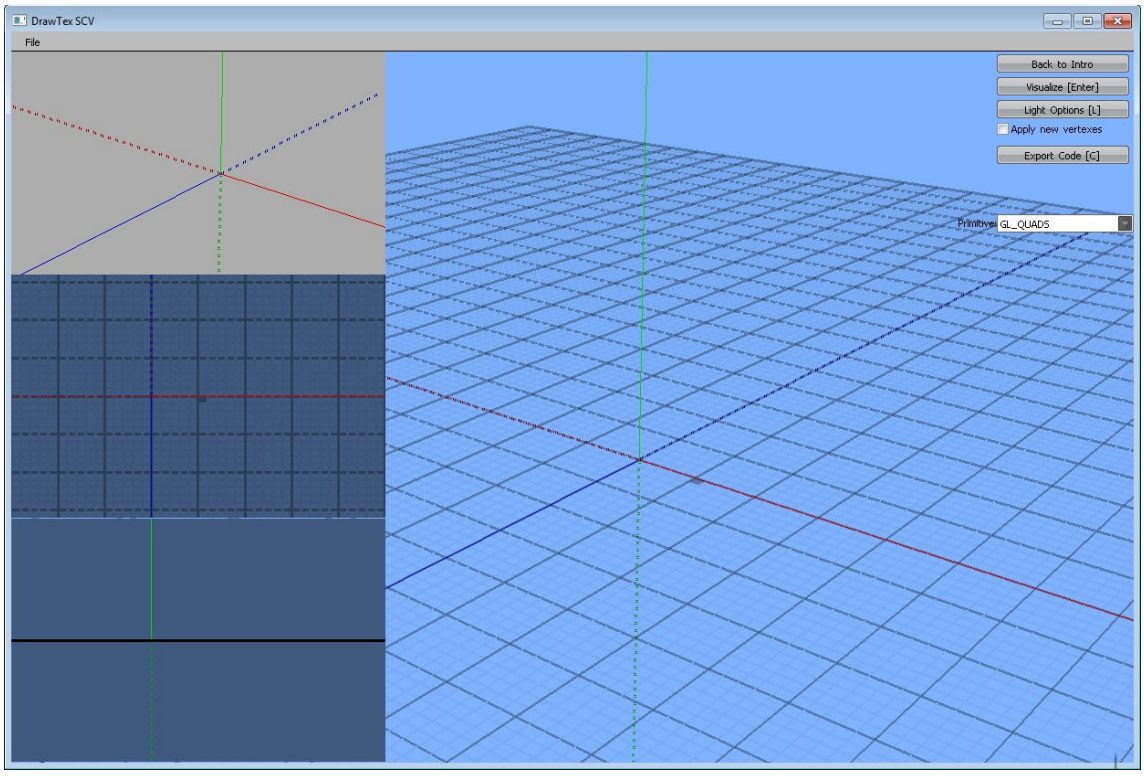

Figura 21 Estado padrão da cena de edição.

Picking é o nome dado às técnicas de seleção de objetos em espaço tridimensional com o uso do *mouse*. A dificuldade da técnica se encontra no mapeamento do espaço tridimensional para um plano - ou seja, no fato de que a renderização torna necessário um mapeamento do plano 2D para o 3D para que o mouse consiga selecionar objetos em profundidade. Como esta é uma tarefa comum em aplicativos de ambiente tridimensional, o OpenGL oferece as ferramentas necessárias para tanto, e de alguns modos diferentes. O modo aplicado no *DrawtexSCV* faz uso das funções fornecidas pela API que possibilitam a inversão do *pipeline de renderização*, mostrado na Figura 4, ou seja, recupera as informações guardadas em *buffer* para determinar se houve colisão do mouse com um objeto.

O modo de mais alto nível de realizar esta tarefa fornecidos pelo OpenGL é chamado de *SelectionMode*, e envolve a manipulação de pilhas de objetos (isto é, controle prévio sobre o desenho de primitivas) e apresenta a desvantagem de basear-se em informação de vértice, sem que o processamento e testes de fragmento tenham sido utilizados – assim, objetos que não aparecem renderizados devido a testes de transparência, profundidade ou *stencil*, por exemplo, podem ser selecionados pela técnica [15]. O modo aplicado no DrawtexSCV utiliza os dados do *DepthBuffer*, isto é, o *buffer* do OpenGL que guarda informações de profundidade de cada pixel para obter a coordenada-original *Z* do mouse (sendo X e Y originais as coordenadas do mouse na tela)para então realizar a operação inversa à projeção.

Um trecho de código com parte fundamental do método de *picking* aplicado no *DrawtexSCV* pode ser visto na Figura 22. Antes da discussão aprofundada sobre esta implementação, no entanto, cabe lembrar alguns conceitos de projeção em espaço tridimensional. Os mínimos dados necessários para levar os dados de um sistema tridimensional a ser mostrado em um *display* bidimensional consistem em: a posição do observador no espaço (ou a posição relativa de todos os objetos em relação a ele); a direção para a qual está orientado e o modo de conversão (tipicamente chamado de *projeção*). São de interesse para este trabalho as projeções ortográfica e perspectiva; e apenas a última no que diz respeito ao processo de *picking* implementado.

```
Vector3f EditorSceneCanvas::mousePick(int mousex, int mousey, bool pointexesDrawn){
   activeCamera->visParameters();
   activeCamera->observerParameters();
   GLint viewport[4];
   GLdouble modelview[16];
   GLdouble projection[16];
   GLdouble posX, posY, posZ;
   glGetDoublev( GL_PROJECTION_MATRIX, projection );
   glGetDoublev( GL_MODELVIEW_MATRIX, modelview );
   glGetIntegerv(GL_VIEWPORT, viewport);
   GLfloat winX = (float)mousex;
   GLfloat winY = (float)(viewport[3]-mousey);
   GLfloat winZ;
   glClear(GL_DEPTH_BUFFER_BIT);
   glEnable(GL_DEPTH_TEST);
   if(!pointexesDrawn){
       drawGround();
   }else{
      int tempGuides = guidesLevel;
      guidesLevel = no_guides;
      drawPointexes();
      guidesLevel = tempGuides;
   }
   glReadPixels((int)winX, int(winY), 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &winZ );
   gluUnProject((double)winX, (double)winY, winZ, modelview, projection, viewport, &posX, &posY,
                                                                                    &posZ);
   return Vector3f((float)posX,(float)posY,(float)posZ);
}
```

Figura 22 Trecho de código contendo o método que implementa parte fundamental do método de *picking*.

Os três conceitos apresentados dizem respeito, no *DrawtexSCV*, à classe **CAMERAGL**, que controla a posição do observador, direção da observação e parâmetros de projeção da cena. Esta classe segue o modelo de câmera virtual no qual a câmera é fixa no eixo de coordenadas Z, orientada para o eixo *Z* negativo, e os objetos da cena são movidos ao redor dela, em transformação inversa ao que seria necessário se os objetos fossem estáticos e a câmera se movesse pelo espaço. Ambos os modelos são equivalentes em resultado e não apresentam diferença para os propósitos deste trabalho. Por definição de interface, o DrawtexSCV opera apenas com a projeção em perspectiva parametrizada na cena e edição. Assim, os parâmetros de ângulo de abertura da câmera, distância dos planos *near* e *far* e aspecto do plano de projeção são dependentes do usuário, embora sejam oferecidos valores padrão.

É por peculiaridade do SCV que são necessárias as chamadas a visParameters e observerParameters da câmera ativa no início do método. Isso se deve ao fato de que mousePick é chamada como resultado de uma interação de mouse, e portanto, parte de uma *callback*. O SCV, no entanto, retém o controle do estado do OpenGL entre uma renderização e outra e assim, quando o usuário clica

no espaço 3D e o método mousePick é chamado, todos os parâmetros descritos acima estão de acordo com o necessário para que o SCV desenhe os componentes de GUI em tela. Após estas chamadas, são utilizadas chamadas fornecidas pelo OpenGL para obter os dados da matriz de *projection,* da matriz de *modelview* e dados da *viewport*. É também por peculiaridade do SCV que é necessário capturar os dados da *viewport* a cada chamada do método – em casos gerais, bastaria obter os dados da *viewport* na inicialização do programa e a cada redimensionamento de janela.

O método então limpa o conteúdo do *DepthBuffer* (isto é, seta a informação de todos os pixels para informar que têm profundidade idêntica, e igual a zero) e procede para escrever os dados de profundidade dos objetos da cena como se eles fossem ser renderizados. O método glReadPixels é então chamado para obter a componente de profundidade no ponto equivalente ao clique do mouse. De posse dessa informação, a posição [*x*,*y*,*z*] do mouse, em espaço 2D, sofre a transformação inversa da projeção. O resultado é um vetor que contém a posição no espaço tridimensional equivalente ao clique do mouse.

O método funciona, pois a leitura da componente de profundidade do pixel é realizada após um falso-processo de desenho, no qual o teste de profundidade do próprio *pipeline* do OpenGL é ativado. Este falso processo de desenho é parametrizado, ainda, de dois modos: se necessário selecionar um dos **POINTEXS** desenhados no espaço; nada é adicionado à cena. No entanto, se o propósito é definir a posição inicial de um novo **POINTEX**, um grande plano *XZ* é desenhado em *Y*=0. Assim, quando o usuário toma um novo ponto para adição, ele tem a coordenada *Y* em zero, e o movimento do mouse o desloca nesse eixo. O *picking* é utilizado para três tarefas em especial: determinar a posição do mouse no *grid* do plano *XZ* e tomar a posição inicial de desenho de um novo Pointex; e selecionar Pointexs no espaço, conforme mostrado na Figura 23.

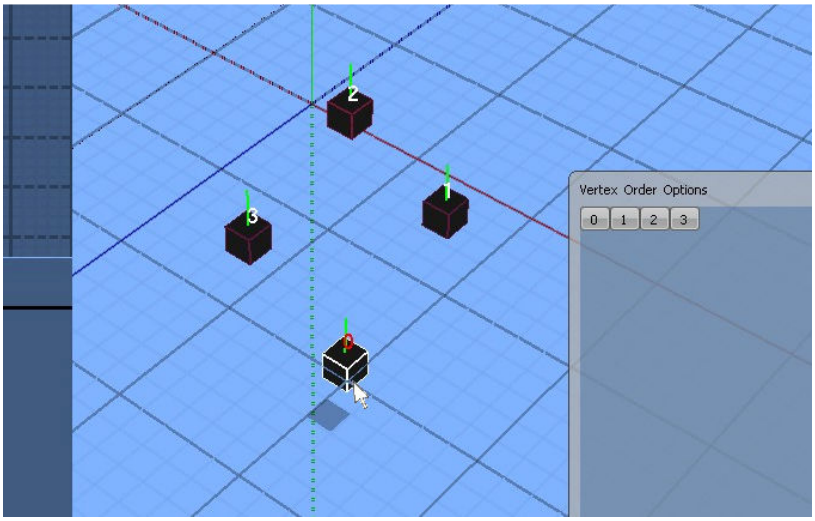

Figura 23 Detalhe da cena de edição, mostrando o mouse selecionando um Pointex com clique.

Além do *Picking*, outro conceito importante e de adoção proveniente dos resultados obtidos com o protótipo foi o uso de *portlets* de visualização. Os *portlets* foram incluídos no protótipo tardiamente, quando a versão atual já estava em desenvolvimento, mas verificou-se empiricamente sua utilidade – e portanto, foram incluídos na versão atual. Esta inclusão testemunha a favor da modularidade obtida com a separação do programa em cenas. Os *portlets* em questão configuram-se como **Canvas** especiais de auxílio ao desenho, nos quais a projeção ortogonal é aplicada para preservar o tamanho e posição dos objetos independente de sua distância do observador. Um destes fornece uma visão superior, de cima para baixo, para que possam ser comparados os **Pointexs** existentes com a posição de interesse (tanto a posição do mouse quanto a posição do ponto tomado para adição) no eixo *XZ*. O outro fornece uma visão lateral, de forma que a posição de interesse para comparação é a do eixo *Y*. Ambos são destacados em uso na Figura 24.

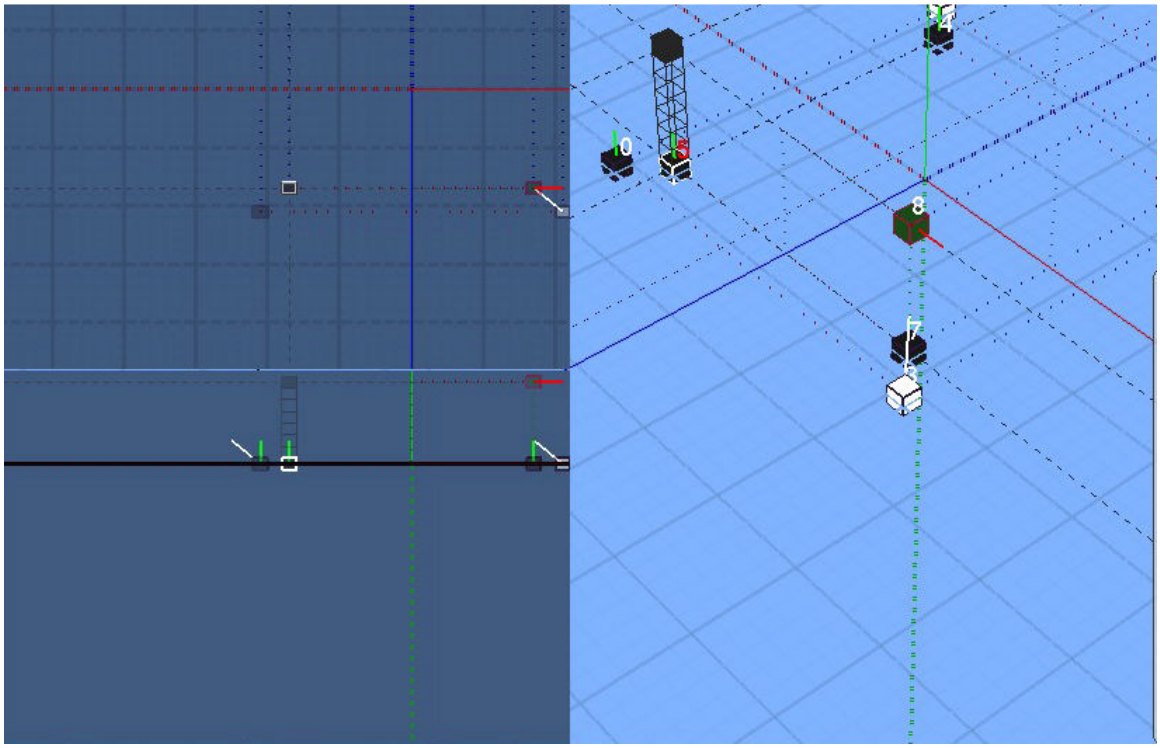

Figura 24 Os portlets de visualização ortogonal sendo utilizados. Pode-se verificar que o Pointex em processo de desenho está alinhado aos demais graças aos portlets, na esquerda da imagem.

Por fim, outro conceitos estritamente ligado ao **EditorScene** é o das guias de desenho e visualização. Foram definidos níveis de guias, variando de zero (**no_guides**) até três (**maximum_guides**); dependendo deste nível são mostradas ou escondidas as diferentes ajudas visuais. Estas guias são implementadas em classes que herdam da classe **Guide** a sua estrutura básica. Como são de composição estática, as Guides utilizam-se da técnica de *display lists* fornecidas pelo OpenGL para otimizar sua renderização. Esta técnica consiste no armazenamento de chamadas (especialmente de desenho) do OpenGL diretamente em placa gráfica para execução posterior, e requer composição estática [16] – usualmente, no início do programa, como é o caso do *DrawtexSCV*. A Figura 25 mostra o efeito do incremento gradativo do nível de guias na interface de edição.

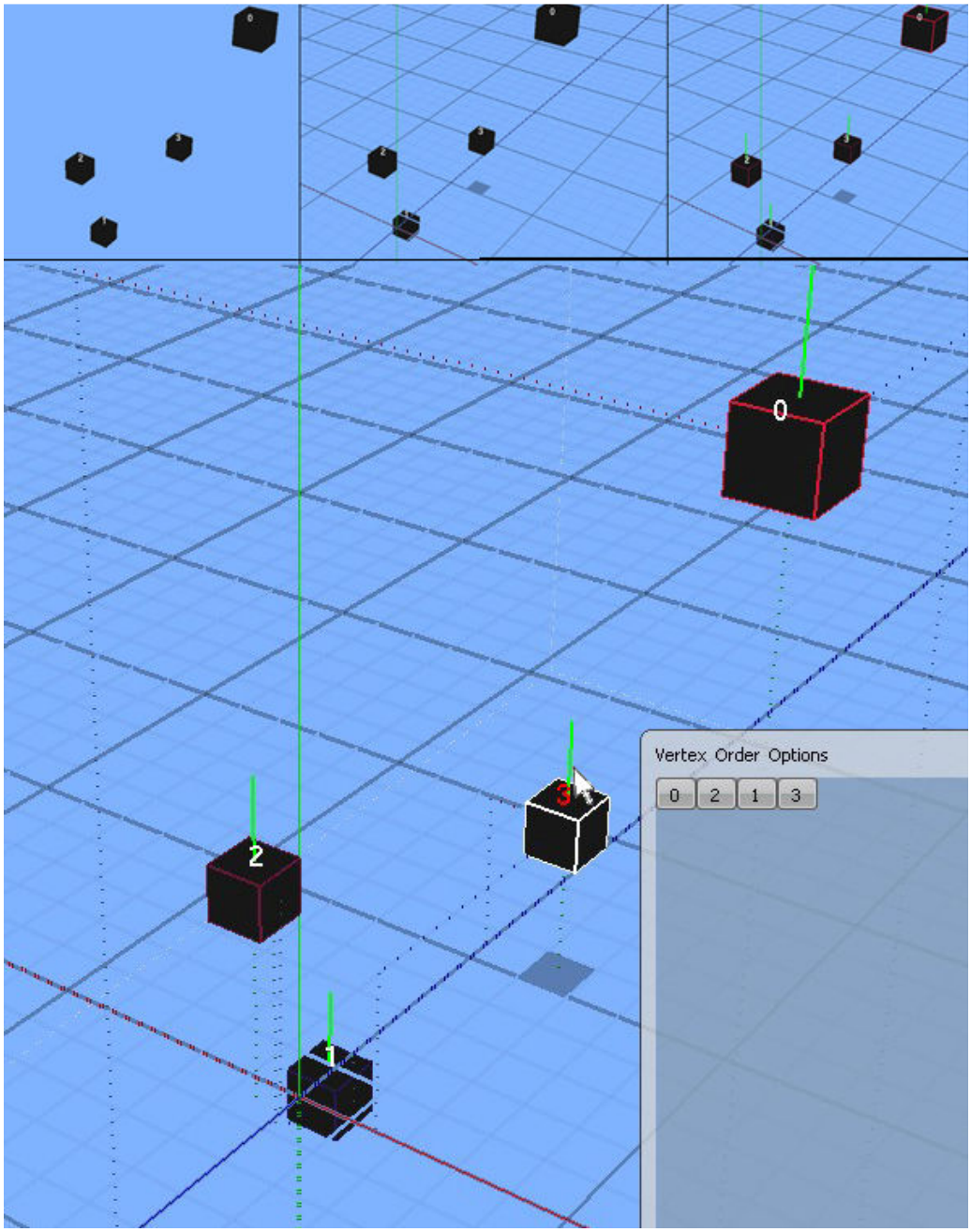

Figura 25 Incremento do nível de guides, com no_guides no topo esquerdo superior e os quadros mostrando a incrementação gradativa, até o maximum_guides na porção inferior da imagem.

É apresentada ainda uma sequência de Figuras, demonstrando a composição de uma primitiva até a sua visualiação. Compõe essa sequências as figuras seguintes, da Figura 26 à Figura 31.

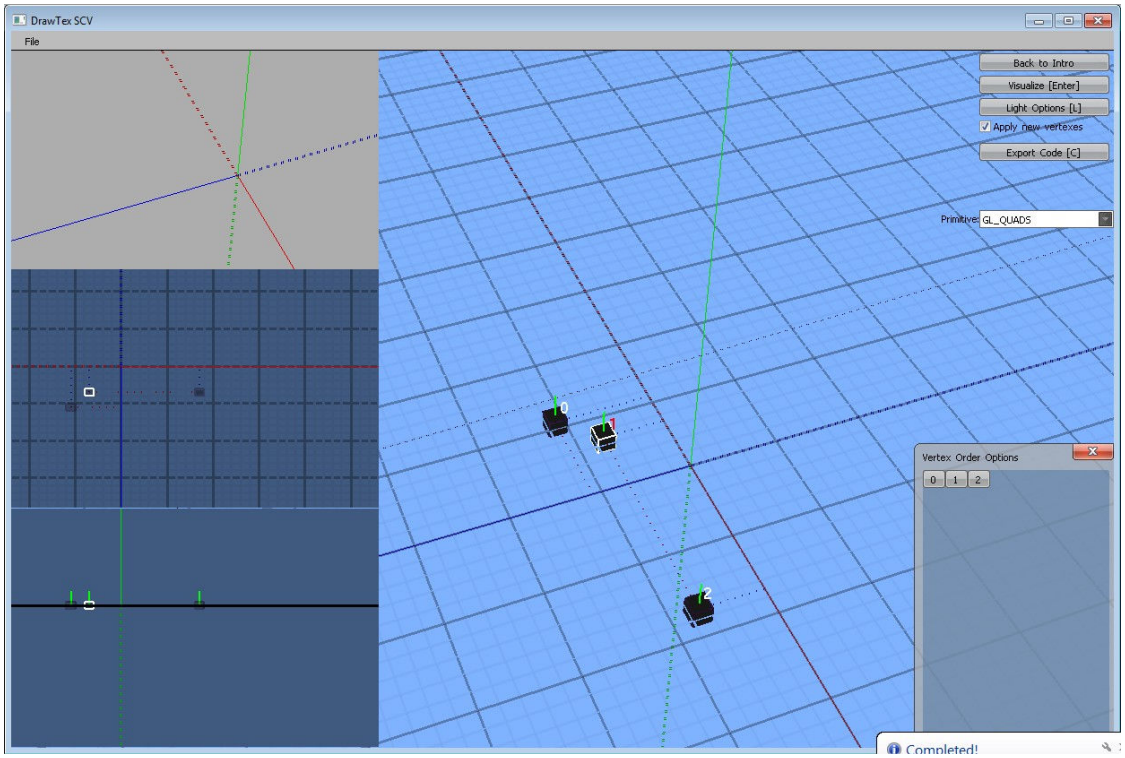

Figura 26 Primeira imagem da sequência da composição de uma primitiva no *DrawtexSCV.*

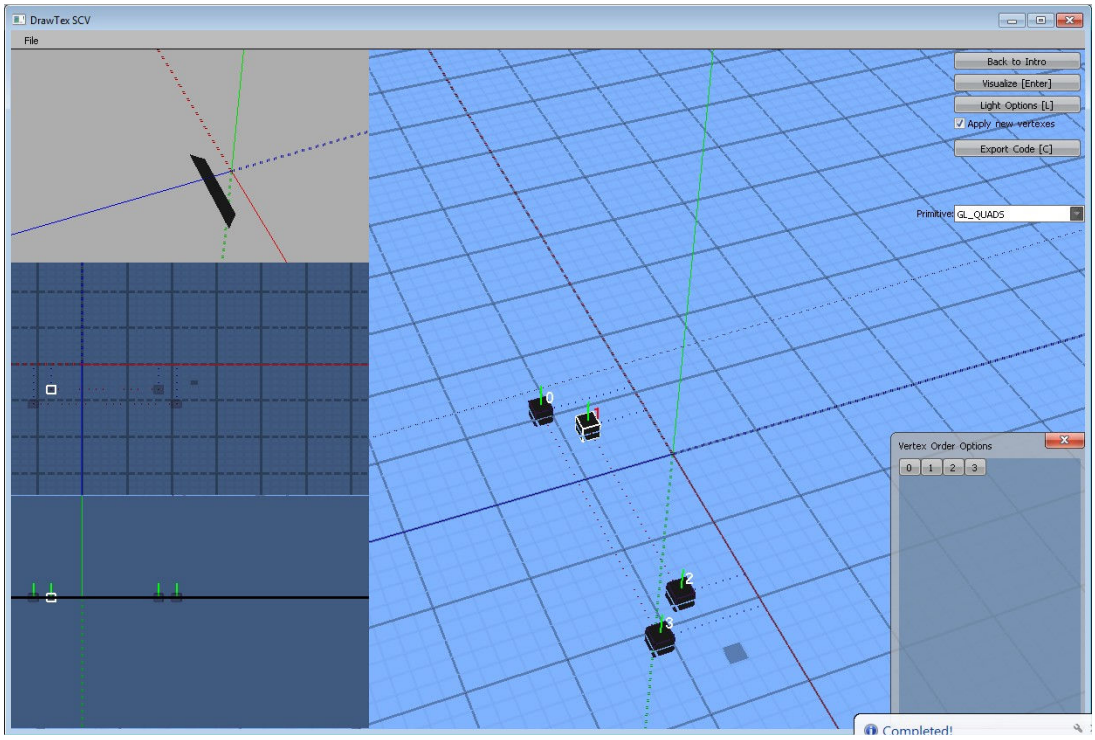

Figura 27 Segunda imagem da sequência da composição de uma primitiva no DrawtexSCV. Nota-se que a partir do quarto ponto a primitiva GL_QUADS já gera resultados na janela de visualização embutida na cena de edição.

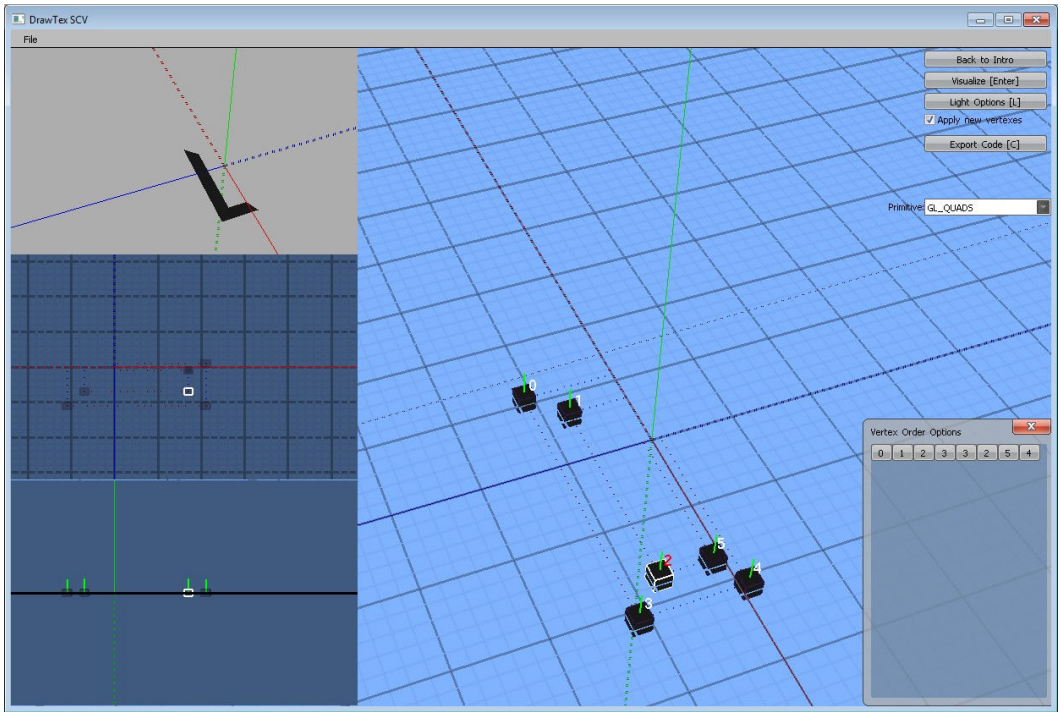

Figura 28 Terceira imagem da sequência da composição de uma primitiva no DrawtexSCV.

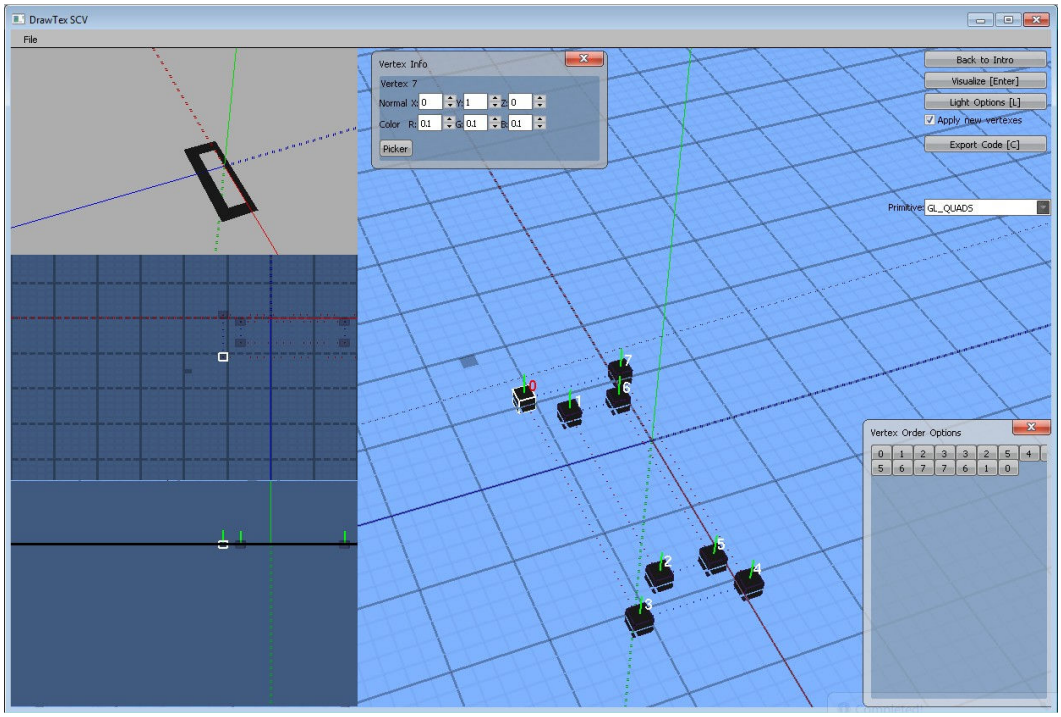

Figura 29 Quarta imagem da sequência da composição de uma primitiva no DrawtexSCV.

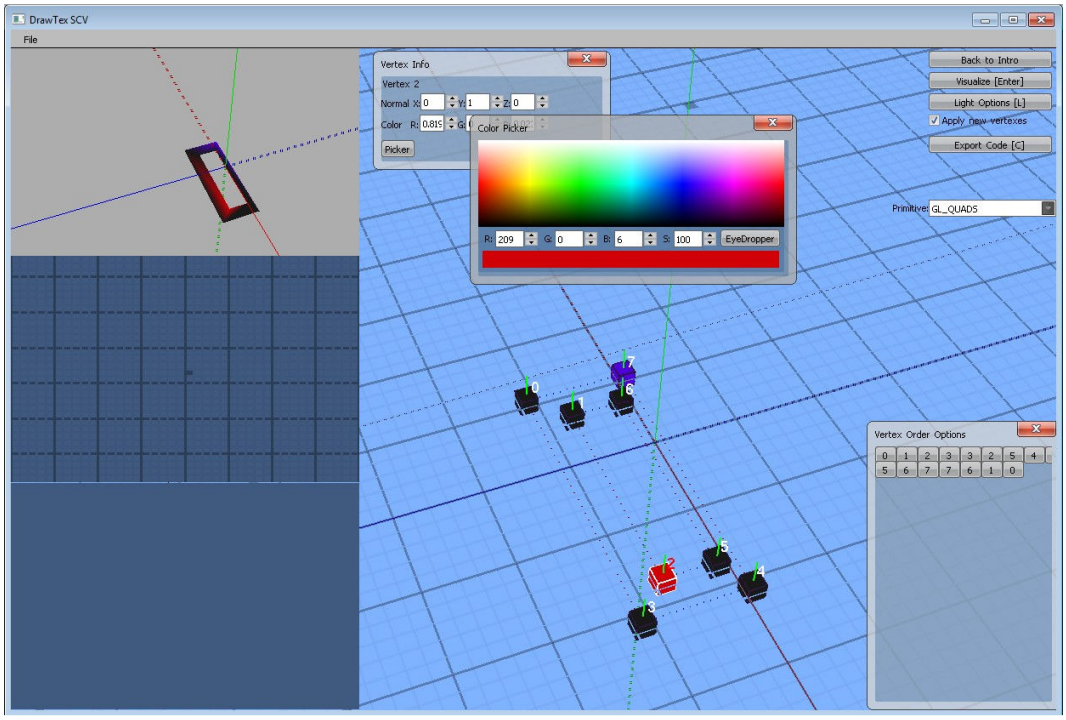

Figura 30 Quinta imagem da sequência da composição de uma primitiva no DrawtexSCV, mostrando a edição das propriedades dos Pointexs.

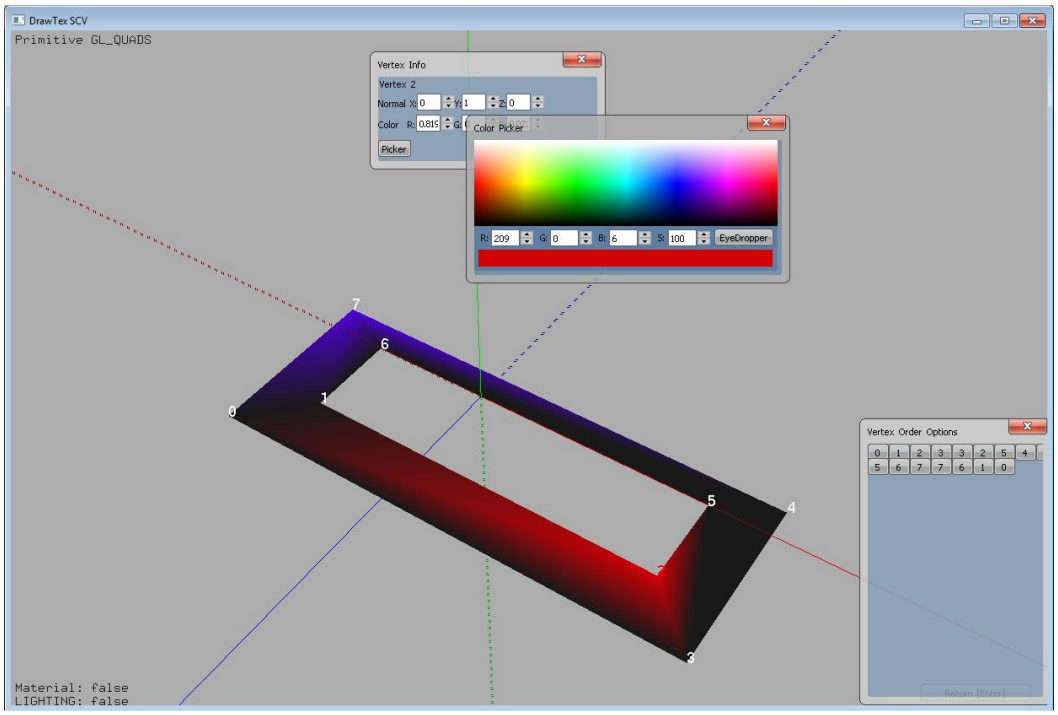

Figura 31 Última imagem da sequência da composição de uma primitiva no DrawtexSCV, demonstrando o resultado na cena de visualização.

4.2.3.2 CodeExport

A cena de exportação está intimamente ligada ao contexto de exportação de código, exposto na seção 4.2.2.3. Como os principais conceitos relacionados já foram explicados na referida seção, cabe apenas relembrar que foi possibilitado, pelo modo de estruturação espelhando as variáreis do OpenGL, traduzir quase diretamente as estruturas de dados do DrawtexSCV em código C/C++. Nota-se que como os parâmetros de exportação não necessariamente coincidem com os de visualização, os **PopUps** são desativados enquanto esta cena encontra-se ativa, para evitar confusão do usuário quanto à interface.

Também é importante especificar a razão pela escolha destas linguagens específicas. O motivo principal é a afinidade do acadêmico realizador do projeto com tais linguagens, o que está intimamente relacionado ao fato de que são as linguagens mais aplicadas na programação de aplicativos de Computação Gráfica. A exportação de código C/C++, além disso, figura como o objetivo do trabalho relacionada às finalidades educativas da ferramenta, visto que estas linguagens são muito utilizadas no meio acadêmico.

A Figura 32 expõe o modo de funcionamento da cena **CodeExport**. Nela, são mostrados os parâmetros fornecidos ao contexto de exportação através dos componentes de interface da cena em questão ao lado de um trecho de código que configura o resultado do processo. Esta pode ser comparada com a Figura 33, na qual os parâmetros foram modificados e pode-se verificar o resultado dessas modificações no código.

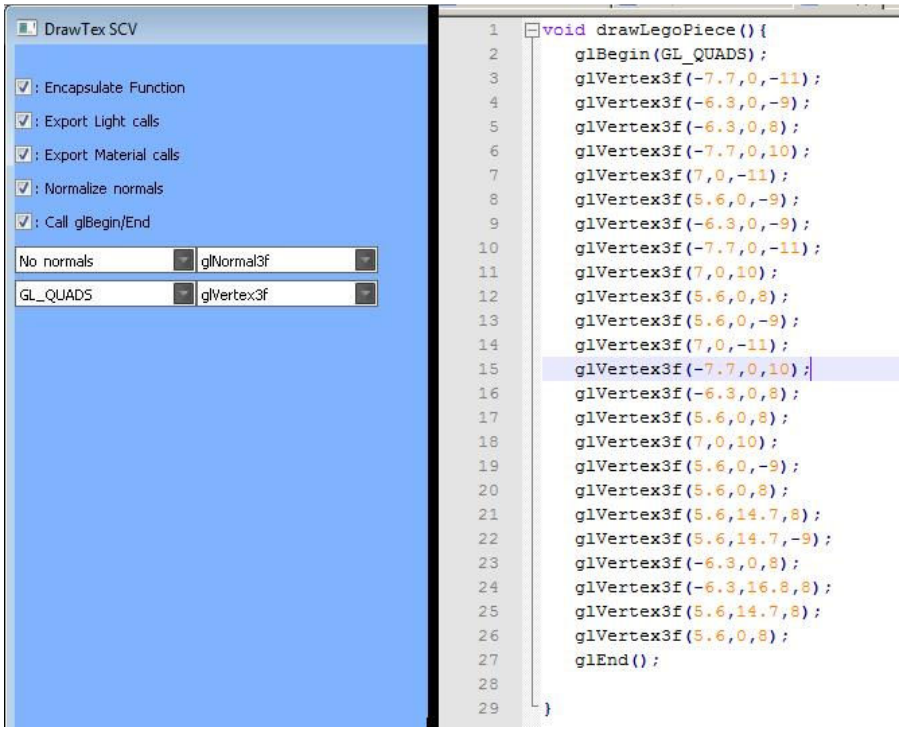

Figura 32 Alguns dos componentes na interface da cena CodeExport, à esquerda, e um trecho do código resultante visualizado em editor de texto.

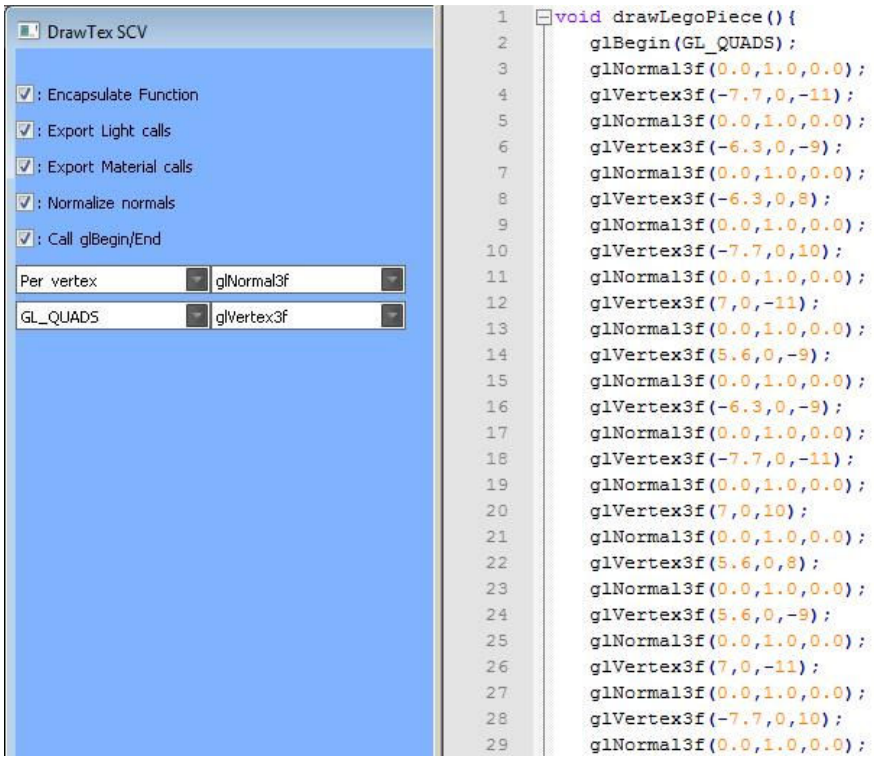

Figura 33 Modificação do atributo "normals" para "Per Vertex" e resultado no trecho de código.

5 RESULTADOS E CONCLUSÕES

Com a utilização da terminologia do OpenGL na interface do programa e o modo de interação baseado na forma como o código OpenGL é produzido (por exemplo, no que diz respeito à ordem de aplicação dos vértices para a composição de primitivas), atingiu-se o objetivo de tornar o software destinado especialmente para o programador OpenGL.

Também houve sucesso no objetivo de modularizar o software para permitir a implementação gradativa e posterior extensão. Isto pode ser verificado pelo fato de que o software foi, de fato, construído desta maneira, e que grande parte das perspectivas futuras de desenvolvimento já se encontram pré-estruturadas no código atual.

Como objetivos de usabilidade, pretendia-se que o usuário pudesse estabelecer em interface tridimensional vértices para a composição de primitivas gráficas. Isto foi alcançado através do uso da técnica de *picking* e da interação entre a cena de edição e os contextos de usuário. Do mesmo modo foi alcançada a meta de permitir múltiplos contextos de edição (a edição de mais de um objeto) em uma mesma sessão de uso. Além disso, permite-se a total parametrização dos atributos de visualização, o que, em conjunto com a visualização em tempo real das edições fornecidas pela cena de visualização e pelo *portlet* de visualização na cena de edição, torna satisfeito o objetivo de que o usuário possa perceber os efeitos das mudanças de atributos do OpenGL na renderização do modelo produzido.

Quanto a permitir a exportação de código parametrizável de forma que o modelo produzido no software possa ser facilmente utilizado e editado em outras aplicações OpenGL construídas em C++, isto foi atingido com a implementação da cena de exportação de código e sua integração com o contexto de usuário referente às opções de exportação. Já o objetivo estabelecer um modo de carregamento de preferências do usuário a partir de arquivo de configurações foi satisfeito com a implementação dos método de inicialização do **PROGRAM**.

Quanto à validação da API SCV, pôde-se concluir que a ferramenta está estruturada de forma a permitir grandes e complexas aplicações utilizando seu sistema de componentes de interface – o *DrawtexSCV*, por exemplo, atingiu a marca de cinco mil linhas de código bruto (sem linhas em branco, de comentários ou só com caracteres de marcação). Algumas das dificuldades enfrentadas com o uso do

SCV devem-se ao fato de que nesta implementação em particular foi necessário, devido à natureza do programa, espelhar os estados internos do OpenGL em dados de aplicação. Isto tornou-se complicado na medida em que, como uma API, o SCV encarrega-se de abstrair alguns processos de baixo nível para o programador. Nenhum impedimento foi encontrado, no entanto, apesar de terem sido necessárias diversas tentativa de implementação em alguns casos.

Por fim, o software será disponibilizado, após testes e possível reestruturação, como ferramenta educativa de uso livre para os acadêmicos de Ciência da Computação da Universidade Federal de Santa Maria, através do site do acadêmico realizar do projeto, satisfazendo assim o último requisito.

No presente momento, o *DrawtexSCV* permite que o usuário crie e edite vértices, suas normais e cores e aplicações em primitivas em um espaço tridimensional; e que atue, através de interface gráfica, sobre uma série de parâmetros de visualização e exportação de código. O código exportado pode ser utilizado diretamente em aplicações OpenGL C/C++.

Pretende-se incrementar, como trabalho futuro, a capacidade de composição de primitivas do DrawtexSCV, de forma que seja possível utilizar combinações de modelos pré-definidos e salvos pelo usuário em bibliotecas; permitindo assim um aumento considerável na complexidade dos modelos produzidos pelo software. Além disso, um módulo de importação de código para a representação interna do DrawtexSCV forneceria funcionalidades interessantes e figura como possibilidade. Por fim, a expansão natural do programa é a inclusão de mais parâmetros e variáveis de estado do OpenGL como parte dos contextos do usuário, até que todas estejam presentes no software. Pode-se também buscar analisar o software sob o ponto de vista do desempenho computacional e, a partir dos resultados, reorganizá-lo em uma segunda versão, utilizando além da técnica de *display lists* os *vertex* e *frame buffer objects* e outras técnicas de otimização do OpenGL.

6 BIBLIOGRAFIA

1. BAUER, N., & FRISCHHOLZ, R. W. (1995). *3-D automatic motion analysis.* Craydon, England: SATA 28th Symposium, Proceedings on Robotics, Motion and Machine Vision in the Automotive Industries, Stuttgart.
2. Brinkmann, R. (1999). *The art and science of digital composing.* San Diego: Morgan Kaufmann.
3. Farin, G., Hoschek, J., & Kim, M.-S. (2002). *Handbook of Computer Aided Geometry Design.* Amsterdam: Elsevier Science.
4. Hearn, D., & Baker, M. P. (1997). *Computer Graphics - C Version.* New Jersey: Prentice Hall.
5. JAHNE, B., HAUBECKER, H., & GEIBLER, P. (1999). *Handbook of Computer Vision and Applications.* Academic Press Systems and Applications, vol. 3.
6. Microsoft. (s.d.). *IRIS GL*. Fonte: Differences between IRIS GL and OpenGL: http://msdn.microsoft.com/en-us/library/dd374238(VS.85).aspx
7. Microsoft. (2010). *IRIS GL and OPENGL*. Fonte: Differences between IRIS GL and OpenGL: http://msdn.microsoft.com/en-us/library/dd374238(VS.85).aspx
8. Pozzer, C. T. (s.d.). Fonte: Animação de personagens: www.inf.ufsm.br/~pozzer
9. Pozzer, C. T. (s.d.). *OpenGL - conceitos básicos.* Fonte: Pozzer: www.inf.ufsm.br/~pozzer
10. Woo, M., Neider, J., Davis, T., & Shreiner, D. (1997). *OpenGL Programming Guide.* Indianapolis: Pearson Education Corporate.
11. Wright, R., & Lipchack, B. (2004). *OpenGL Super Bible.* Indianapolis: Sams Publishing.
12. Woo, Maxon. OpenGL Programming Guide: The Official Guide to Learning OpenGL. 3rd ed. Reading, MA: Addison-Wesly, 1999.