INSTITUTO NACIONAL DE MATEMÁTICA PURA E APLICADA

RELATÓRIO DE INICIAÇÃO CIENTÍFICA

JOÃO LIMA SANTANELLI

ORIENTADOR: PROFESSOR LUIZ VELHO

REALIDADE AUMENTADA MÓVEL

Rio de Janeiro
2012

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Este relatório descreve as atividades realizadas por João Lima Santanelli no Laboratório Visgraf-IMPA durante o período de setembro de 2011 a maio de 2012 sob a supervisão do professor Luiz Velho. O trabalho foi apoiado pelo CNPq mediante o pagamento de uma bolsa de iniciação científica.

1 INTRODUÇÃO

Realidade aumentada pode ser definida como a integração de informações retiradas do mundo real, através de sensores mecânicos ou eletrônicos, com elementos virtuais, gerados por processos computacionais.

Um aplicativo de realidade aumentada pode dispor de informações dos mais diversos tipos de sensores, como câmeras, acelerômetros e bússolas, entre outros.

Quanto maior for a quantidade de informação captada pelos sensores, mais elevado será o grau de verossimilhança do aplicativo, permitindo uma interação mais agradável entre o usuário e o programa.

1.1 MOTIVAÇÃO

Com o desenvolvimento de novas tecnologias de *hardware*, principalmente na área de dispositivos móveis para comunicação, torna-se mais fácil a implementação de aplicativos de realidade aumentada para estes dispositivos.

Além disso, a busca pela comodidade leva as pessoas a buscarem meios mais fáceis e intuitivos de se comunicar com o mundo virtual, abrindo espaço para um novo conceito de interação entre usuário e aplicação.

1.2 OBJETIVO DO TRABALHO

Desenvolver aplicativos de realidade aumentada para dispositivos móveis, aplicando os conceitos de programação para Android, OpenGL e código nativo estudados.

2 CONCEITOS DE PROGRAMAÇÃO PARA ANDROID

Para iniciar os trabalhos de desenvolvimento de aplicativos de realidade aumentada móvel, foi necessário estudar, primeiramente, os conceitos básicos da programação voltada para o sistema operacional Android.

As aplicações para Android são divididas em quatro tipos de componentes: *Activities*, *Services*, *Content Providers* e *Broadcast Receivers*.

2.1 ACTIVITIES

As *Activities* representam as interfaces do aplicativo com o usuário. Um programa destinado à leitura de *e-mails*, por exemplo, apresenta uma *Activity* que mostra a lista de *e-mails* recebidos, na qual o usuário pode escolher qual deseja ler, e outra na qual a mensagem do *e-mail* escolhido é exibida. Na FIG.2.1, pode-se observar o ciclo de vida de uma *Activity*.

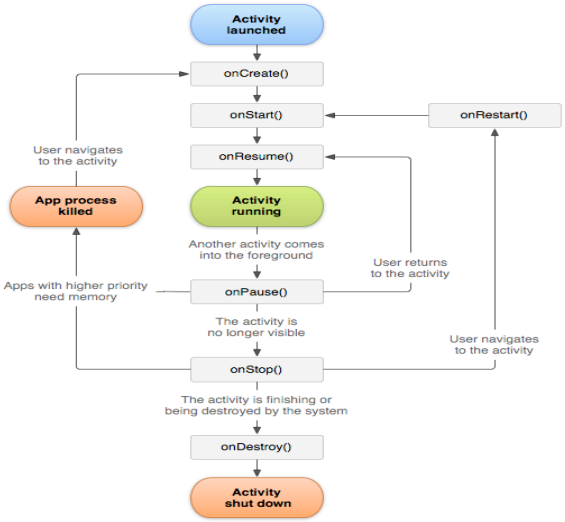

FIG.2.1 Ciclo de vida de uma *Activity*.

2.2 SERVICES

Os *Services* são componentes executados em segundo plano, não apresentando nenhuma interface com o usuário. Um *Service* pode, por exemplo, executar uma música enquanto o usuário controla os movimentos de um personagem de um jogo.

2.3 CONTENT PROVIDERS

Content Providers são classes responsáveis por gerenciar o conjunto de dados compartilhados das aplicações. Permitem salvar o estado de uma aplicação que deverá ser interrompida.

2.4 BROADCAST RECEIVERS

Os *Broadcast Receivers* são os componentes responsáveis por detectar requisições do sistema por determinado tipo de serviço.

2.5 APLICATIVO DESENVOLVIDO

Para consolidar os conceitos estudados sobre programação para *Android*, foram desenvolvidos alguns aplicativos de teste. O mais significativo dentre eles foi o protótipo de um quebra-cabeça numérico, através do qual foi possível visualizar o funcionamento de uma *Activity*, bem como explorar os recursos da classe *View*, que fornece elementos para a criação da interface com o usuário. Na FIG.2.2, é mostrada a interface do protótipo.

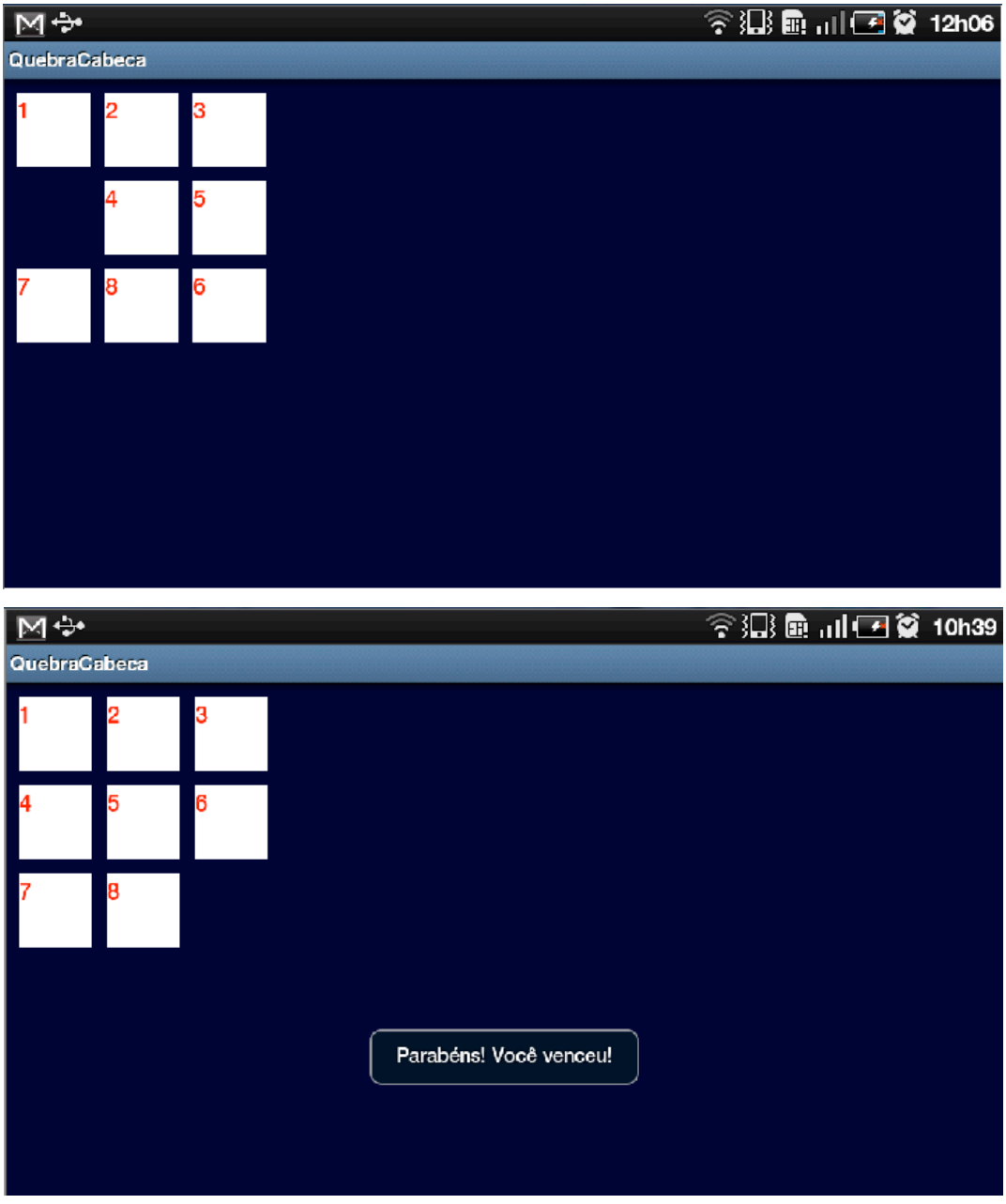

FIG.2.2 Protótipo de um quebra-cabeça.

3 CONCEITOS DE OPENGL

A OpenGL é um conjunto de funções que fornecem acesso aos recursos do *hardware* de vídeo de um dispositivo. Para dispositivos móveis, que, geralmente apresentam limitações tanto em memória quanto em recursos gráficos, é recomendável que se utilize uma versão reduzida desta biblioteca, a OpenGL ES. Foi escolhida a versão 2.0 para o desenvolvimento deste trabalho, pois esta apresenta *pipeline* de renderização programável, conferindo maior liberdade na implementação de efeitos gráficos especiais. Serão abordados, a seguir, alguns conceitos sobre a OpenGL ES 2.0.

3.1 CÂMERA VIRTUAL

Um conceito importantíssimo para se trabalhar com a construção de elementos gráficos utilizando OpenGL é o de câmera virtual. Ela simula o olho do observador, projetando toda a região visível, contida em um tronco de pirâmide, em um anteparo bidimensional. Na FIG.3.1, pode-se observar o esquema de projeção da cena pela câmera virtual.

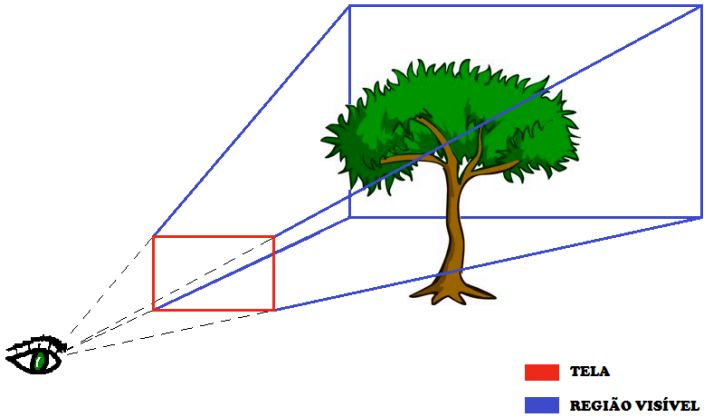

FIG.3.1 Câmera virtual e a projeção da cena.

Seu posicionamento na cena é definido por três vetores: *eye*, *center* e *up*.

O vetor *eye* indica o local onde estaria situado o olho do observador. *Center* representa o centro da cena, ou seja, para onde o observador está olhando. Por fim, o vetor *up* indica a rotação da câmera em relação ao eixo definido pelo olho do observador e o centro da cena, que pode ser obtido através de uma operação de subtração vetorial envolvendo os vetores *eye* e *center*. Na FIG.3.2, observa-se a disposição destes vetores.

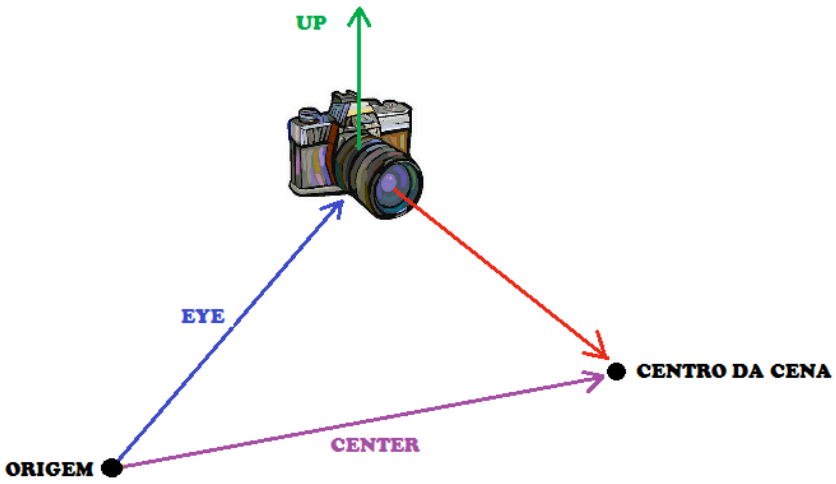

FIG.3.2 Posicionamento da câmera virtual na cena

3.2 PIPELINE DE RENDERIZAÇÃO

Renderização é o nome dado ao processo de transformação dos dados que representam a cena tridimensional em uma figura bidimensional que será exibida na tela do dispositivo. Esse processo funciona basicamente como uma linha de montagem, em que, ao longo de cada etapa, são realizadas operações específicas sobre os dados de entrada. A FIG.3.3 (MUNSHI, 2008, p.4), mostra, esquematicamente, as etapas do *pipeline* de renderização, que serão abordadas a seguir.

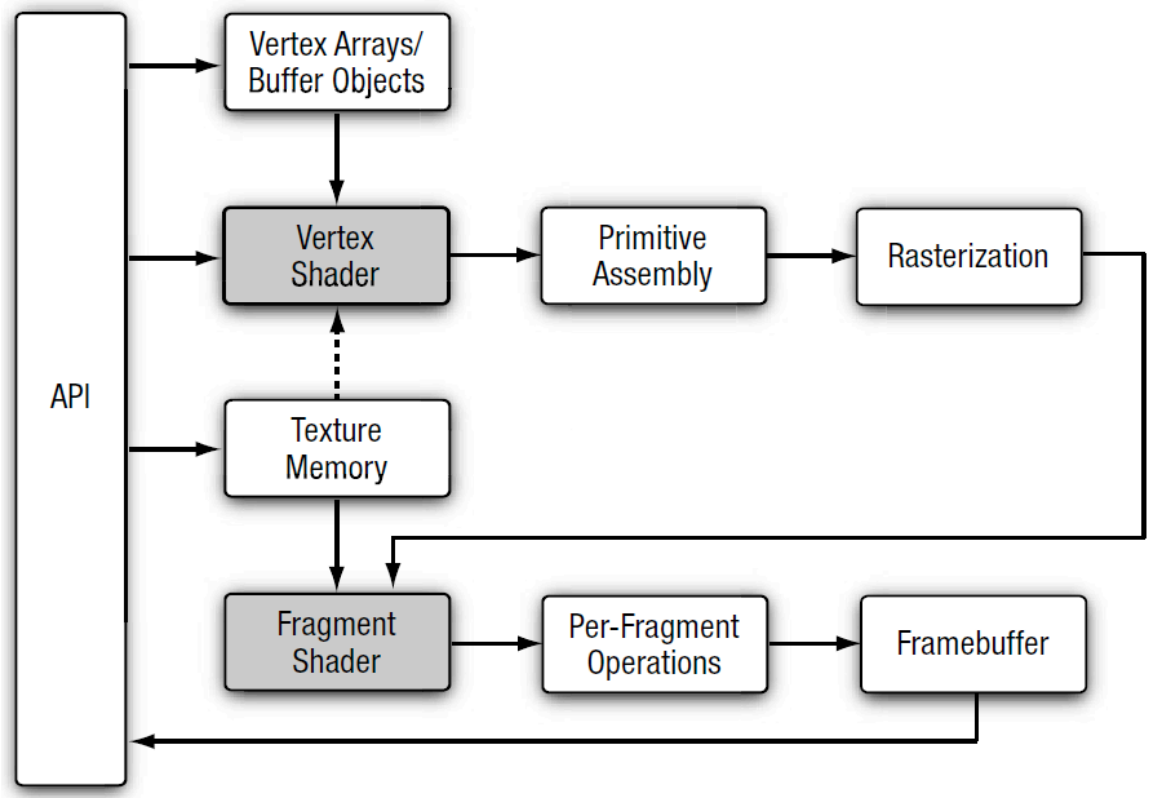

FIG.3.3 *Pipeline* de renderização

3.2.1 VERTEX ARRAYS E BUFFER OBJECTS

Os *Vertex Arrays* são estruturas que permitem o armazenamento dos dados relativos a cada vértice de forma arbitrária pelo programador. Cada vértice pode ter tantos atributos quantos forem necessários, basta que, para isso, seja realizado o devido tratamento no *Vertex Shader* (que será discutido na subseção 3.2.2).

Em um *Vertex Array*, podem ser armazenados atributos como, por exemplo, posição, cor, vetor normal e coordenadas de textura relativos a cada vértice.

Os *Buffer Objects* servem para realizar o *cacheamento* da memória gráfica, evitando que os dados precisem ser reenviados a cada iteração do *pipeline*.

3.2.2 VERTEX SHADER

O *Vertex Shader* é uma das partes programáveis do *pipeline* de renderização, sendo responsável pelas operações realizadas sobre os vértices anteriormente armazenados nos *Vertex Arrays*.

Como entrada, o *Vertex Shader* pode receber dados relativos aos vértices sob a forma de variáveis do tipo *Attribute* ou do tipo *Unifform*. As primeiras representam as características individuais de cada vértice (geralmente armazenadas nos *Vertex Arrays*) como, por exemplo, posição, cor e vetor normal. Já o segundo tipo representa aquelas características que são comuns a todos os vértices, como, por exemplo, o vetor que representa a direção da luz (caso se deseje implementar efeitos de luz e sombra).

Na FIG.3.4 (MUNSHI, 2008, p.5), pode-se observar, esquematicamente, a estrutura do *Vertex Shader*.

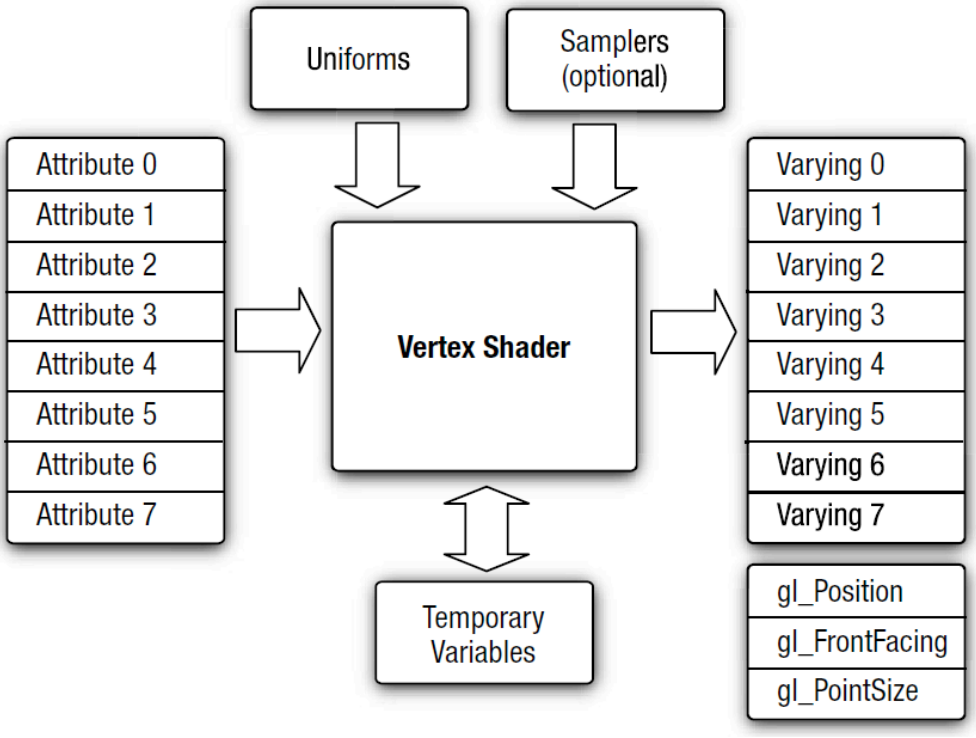

FIG.3.4 *Vertex Shader*.

É possível, também, passar valores diretamente do *Vertex Shader* para o *Fragment Shader* (outra parte programável do *pipeline*, que será abordada na subseção 3.2.5) através das variáveis do tipo *Varying*. Além disso, o *Vertex Shader* apresenta algumas saídas pré-definidas, dentre as quais pode-se destacar a *gl_Position*, que atribui um valor de posição ao vértice dentro da cena.

3.2.3 PRIMITIVE ASSEMBLY

Esta etapa do *pipeline* agrupa os vértices saídos do *Vertex Shader* em primitivas, que podem ser de três tipos: pontos, retas ou triângulos. Estas podem ser ou não descartadas de acordo com sua posição relativa ao tronco de pirâmide que representa a região visível da cena.

Nesta fase, também ocorre a projeção das primitivas no *ViewPort*, que é o anteparo bidimensional onde a câmera virtual projeta a cena observada.

3.2.4 RASTERIZATION

Nesta etapa, as primitivas são convertidas em fragmentos bidimensionais. Estes, após devidamente tratados, darão origem aos *pixels*, que serão projetados no *Framebuffer* (que será abordado na subseção 3.2.7).

3.2.5 FRAGMENT SHADER

É uma das etapas programáveis do *pipeline* de renderização. Nele podem ser efetuadas operações sobre os fragmentos. Podem receber as *Varyings* (citadas na subseção 3.2.2) diretamente do *Vertex Shader*. Ao final das operações, é atribuída uma cor ao fragmento através

da variável de saída *gl_FragColor*. Na FIG.3.5 (MUNSHI, 2008, p.8), observa-se a estrutura do *Fragment Shader*.

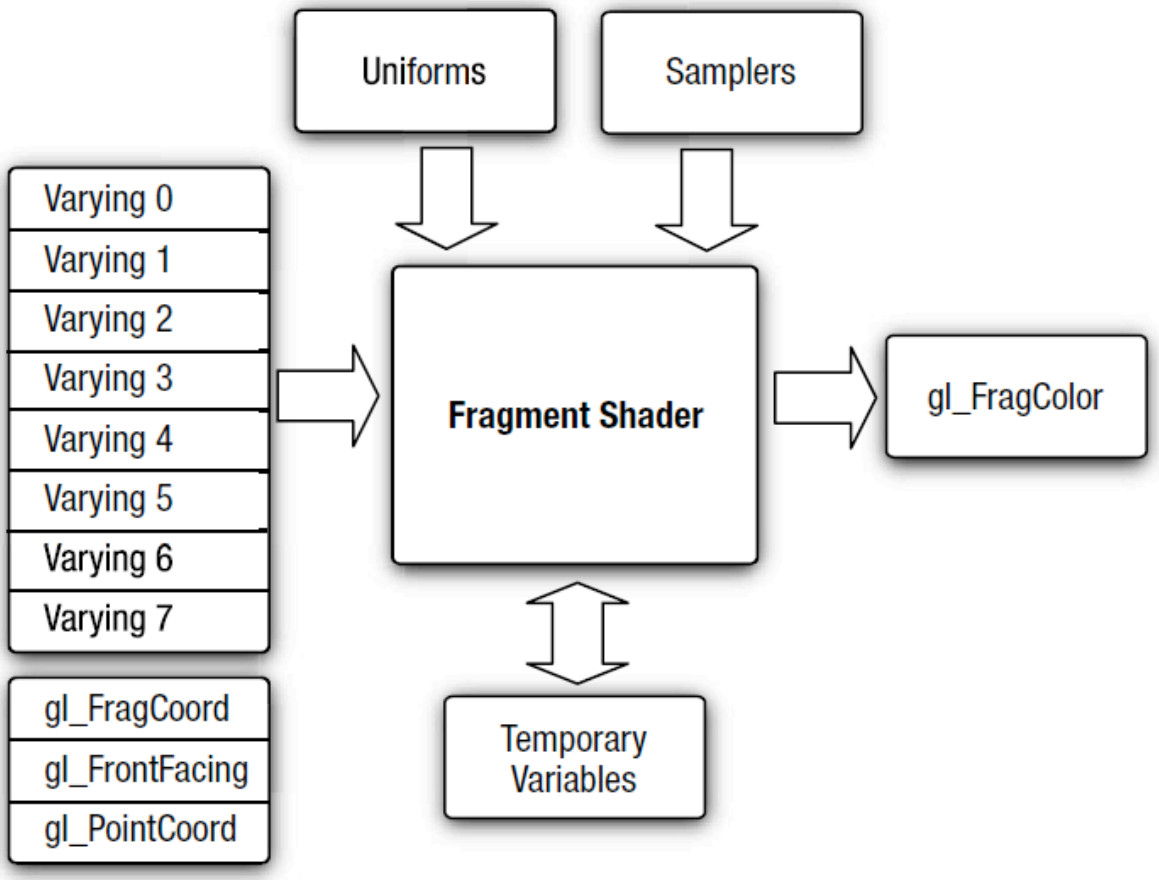

FIG.3.5 *Fragment Shader*.

3.2.6 OPERAÇÕES POR FRAGMENTOS

Consiste em um conjunto de testes que determinam se o fragmento será ou não considerado na geração dos *pixels*.

Dentre estes testes, os principais são o *Scissor Test*, que determina se o fragmento está ou não na região que será exibida, e o *Depth Test*, que determina qual tem menor profundidade, no caso de mais de um fragmento ter sido mapeado no mesmo ponto do *View Port* (citado na subseção 3.2.3).

3.2.7 FRAMEBUFFER

É o *buffer* que armazena os valores dos *pixels* que serão exibidos na tela.

3.3 APLICATIVO DESENVOLVIDO

Para consolidar os conceitos de OpenGL, foi desenvolvido um aplicativo que capta informações do mundo real através do acelerômetro do dispositivo, permitindo que seja conhecida a inclinação do mesmo. Com base nestes dados, o programa modifica a posição da câmera virtual em relação a um objeto virtual, permitindo ao usuário selecionar o ângulo de observação apenas mudando a inclinação do dispositivo. Também é possível girar o objeto virtual através do toque na tela. Na FIG.3.6, observam-se imagens do aplicativo.

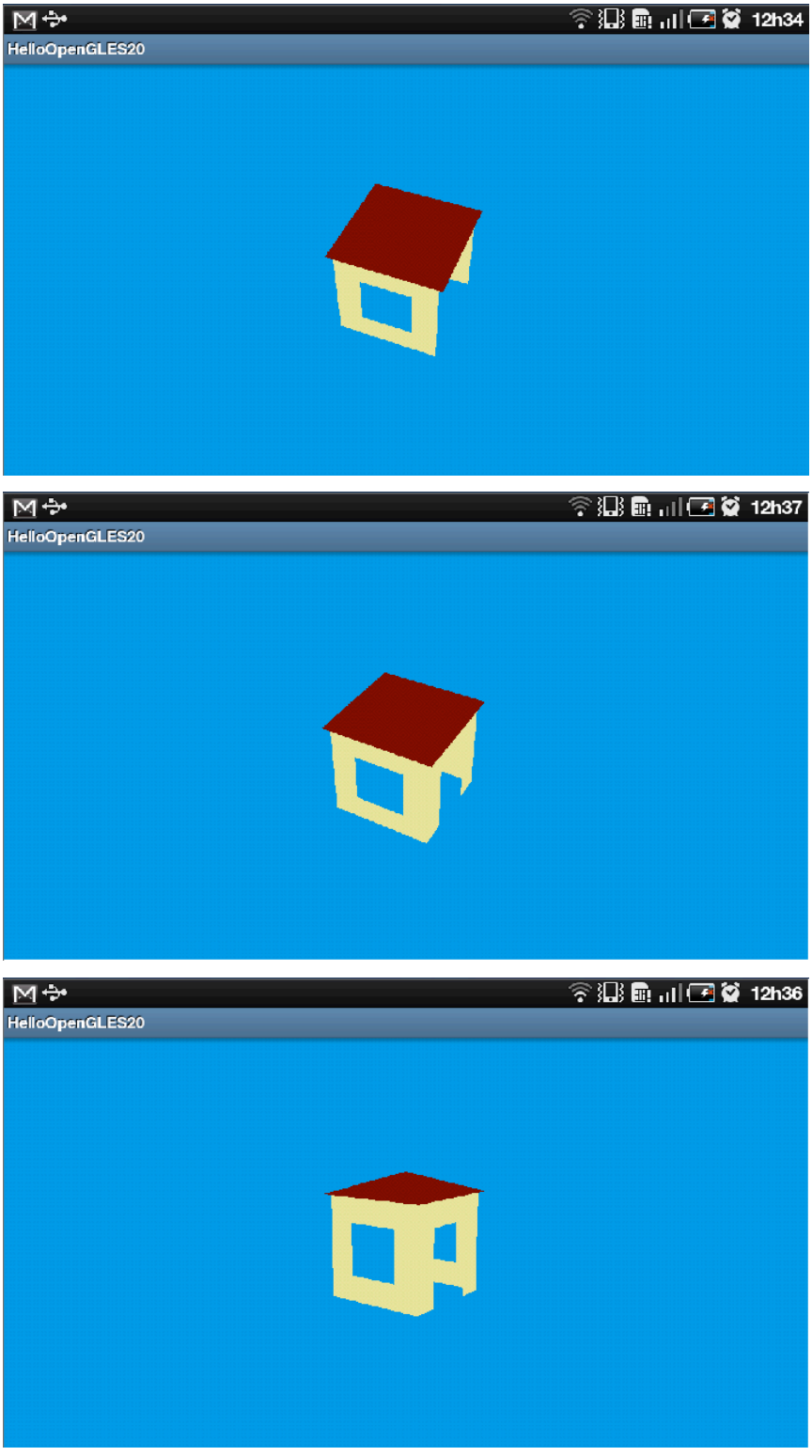

FIG.3.6 Aplicativo usando OpenGL e sensores do dispositivo.

4 ACESSO A FUNÇÕES NATIVAS USANDO JNI

4.1 CÓDIGO NATIVO

Para se garantir a otimização dos recursos computacionais, que geralmente são escassos em dispositivos móveis, a solução adotada foi a utilização de código pré-compilado em C/C++, linguagem nativa do sistema operacional Android.

Para isso, foi utilizada a *Java Native Interface* (JNI), que permite o acesso a funções desenvolvidas em código nativo.

4.2 APLICATIVO DESENVOLVIDO

Para testar o funcionamento desse recurso, foi criado um programa de teste que imprime na tela uma *string* oriunda de uma função nativa. Na FIG.4.1, observa-se o resultado do programa.

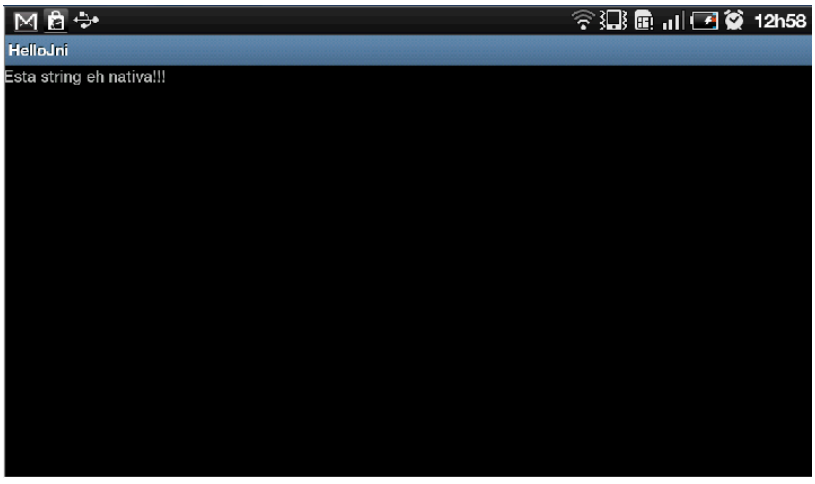

FIG.4.1 *String* oriunda de função nativa

5 APLICATIVO FINAL

Para consolidar os fundamentos aprendidos, foi desenvolvido um aplicativo de teste utilizando conceitos de programação para Android, OpenGL e código nativo.

O aplicativo consiste em gerar recursivamente um sólido a partir de um tetraedro. De acordo com o nível de detalhamento escolhido pelo usuário, cada face do sólido anterior se divide em três novas faces através da inserção de um novo vértice em seu centro. O novo vértice é, então, deslocado de forma a se situar na superfície de uma esfera imaginária. Repetindo o processo várias vezes, o sólido vai-se aproximando de uma esfera. É permitido também ao usuário selecionar o nível de complexidade da textura do plano de fundo (*Background*). Esta textura consiste em uma malha semelhante a um tabuleiro de xadrez, tendo seu número de casas aumentado de acordo com o nível escolhido pelo usuário. Cada casa representa um *pixel*. O tamanho da textura varia desde 64 *pixels* (nível 0) até 2^{16} *pixels* (nível 5), sendo multiplicado por quatro a cada nível. Da FIG.5.1 até a FIG.5.6 observam-se algumas imagens do programa.

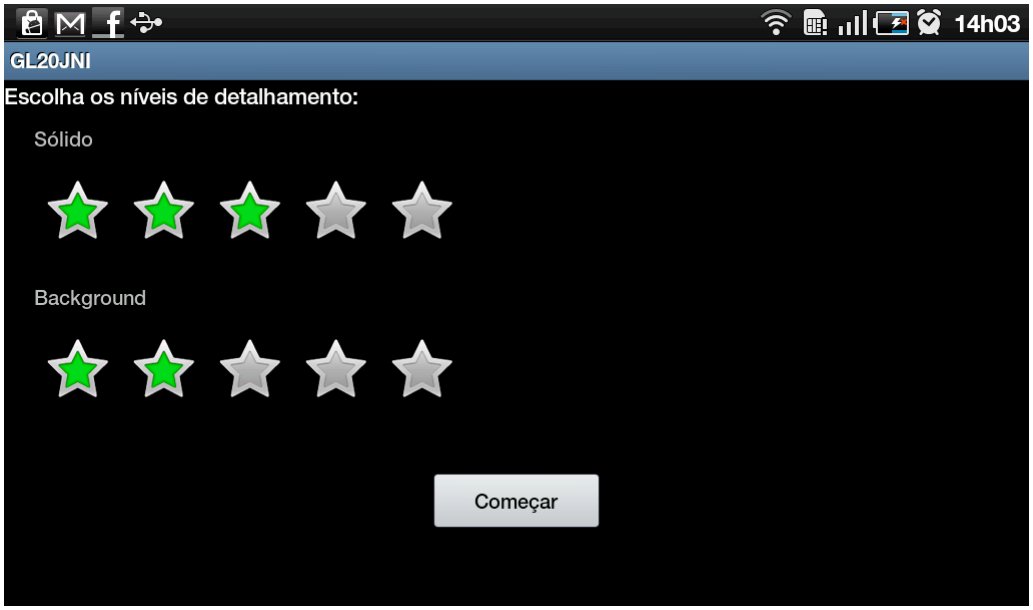

FIG.5.1 Interface inicial.

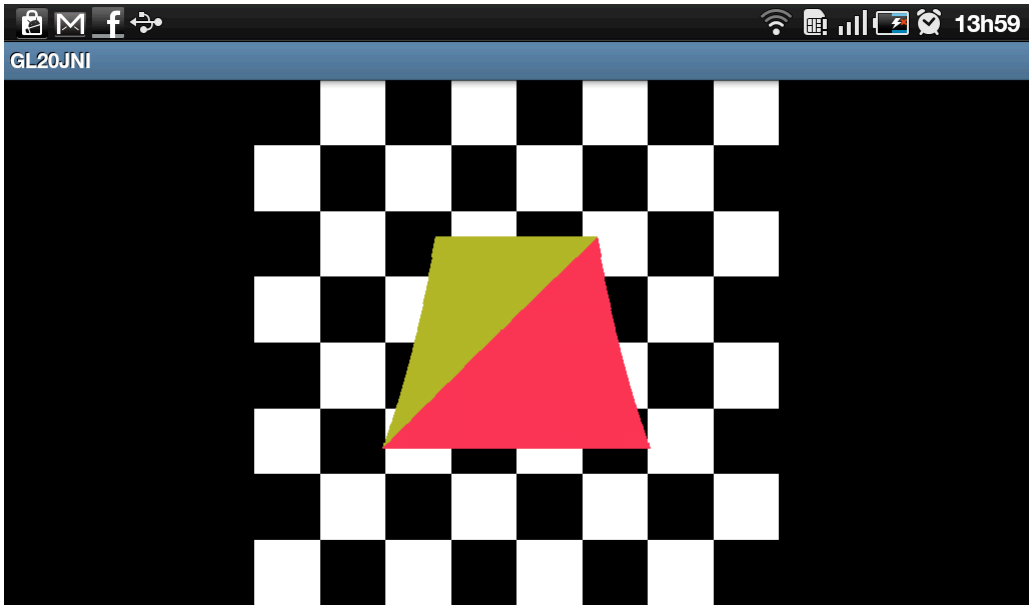

FIG.5.2 Sólido e *Background* no nível zero.

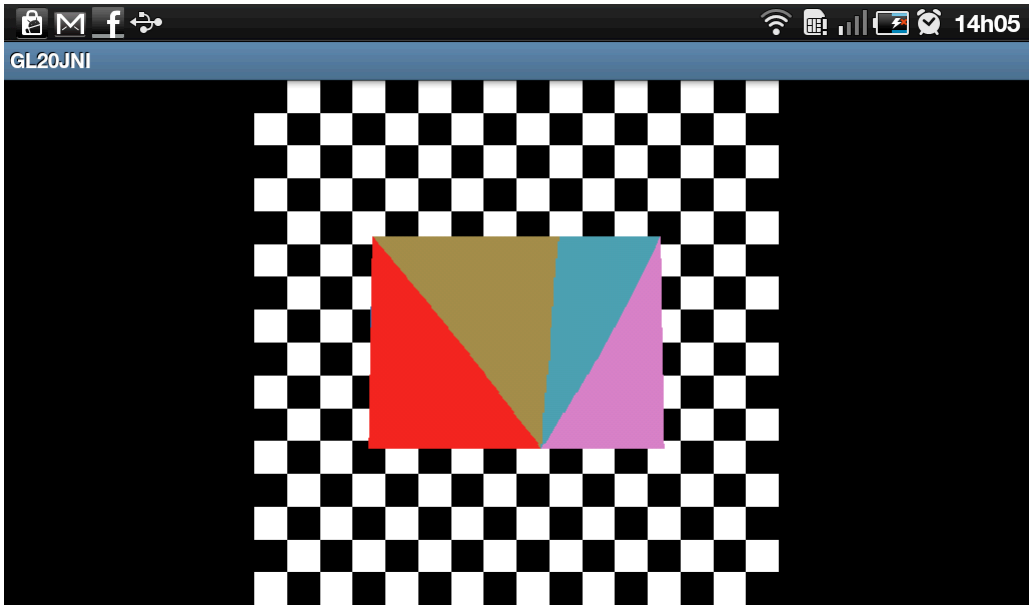

FIG.5.3 Sólido e *Background* no nível um.

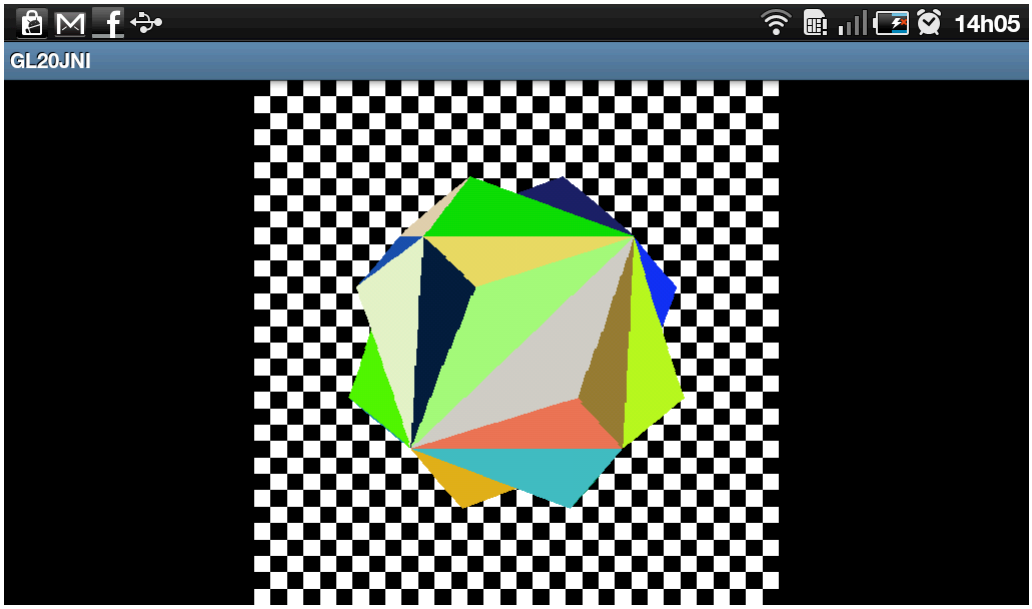

FIG.5.4 Sólido e *Background* no nível dois.

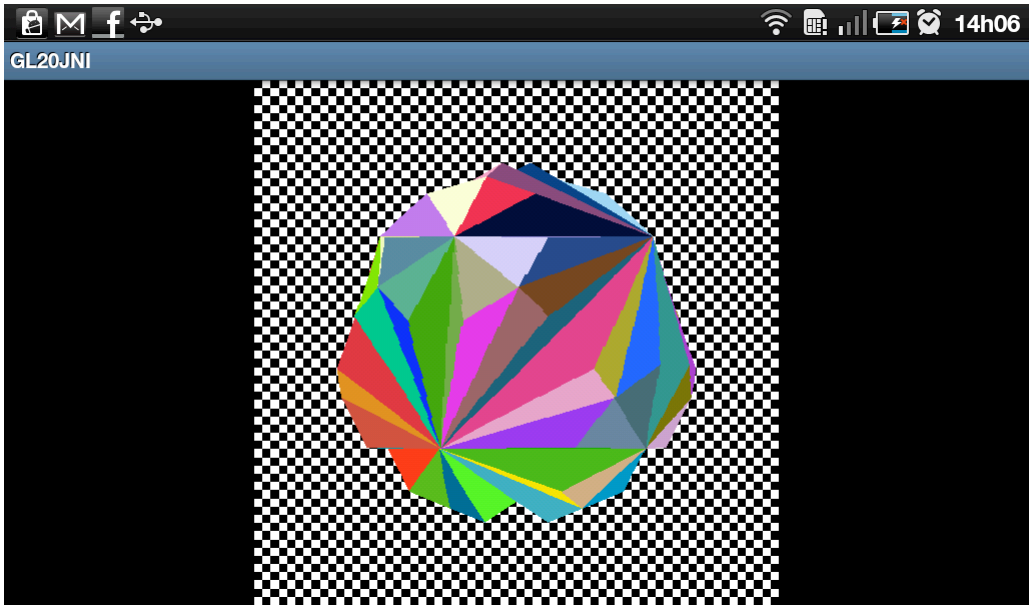

FIG.5.5 Sólido e *Background* no nível três.

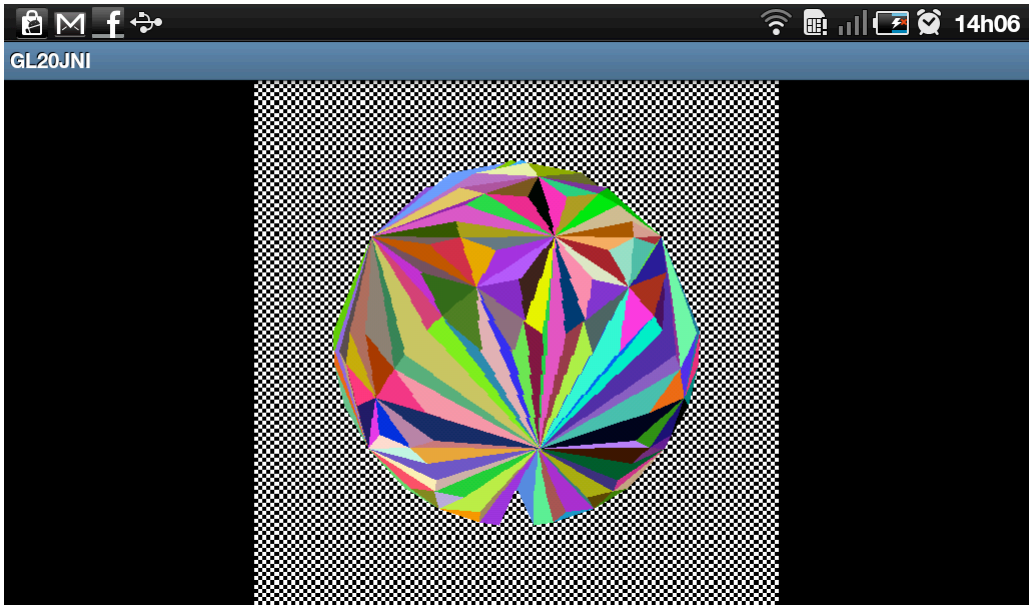

FIG.5.6 Sólido e *Background* no nível quatro.

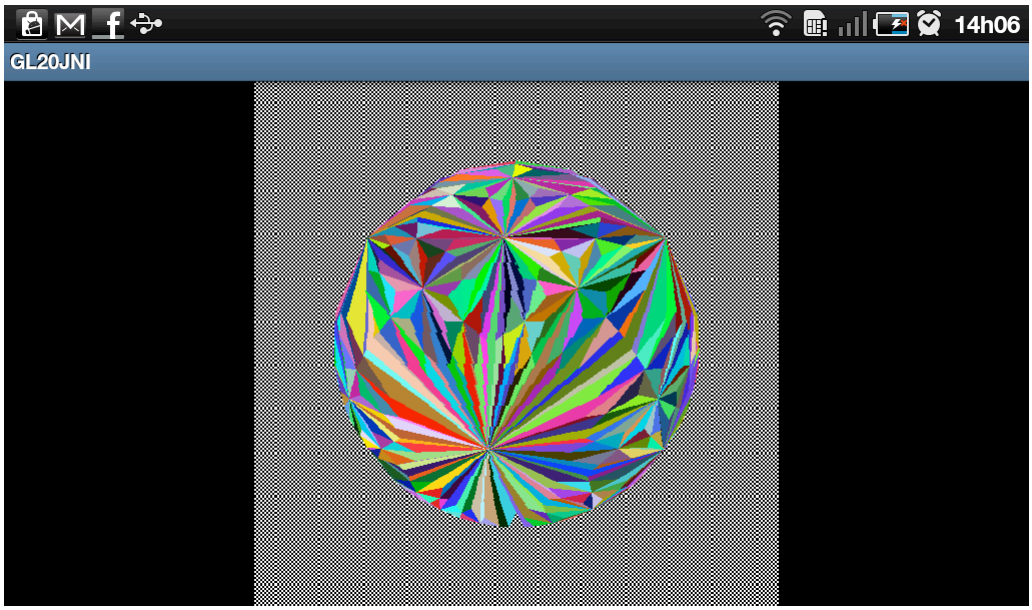

FIG.5.7 Sólido e *Background* no nível cinco.

Foi observado que a captura das imagens do programa foram prejudicadas pela redução de seu tamanho original, de modo que, na tela do dispositivo móvel utilizado para teste, as distorções na textura do plano de fundo, observadas na FIG.5.6 e na FIG.5.7, não foram tão significativas.

Foram testados os limites do programa através do aumento do número de faces do sólido e do aumento do número de *pixels* da textura do *Background*. Observou-se que, independentemente da quantidade de faces do sólido, o tamanho máximo da imagem de fundo suportado pelo aplicativo foi de 2^{18} *pixels*, como pode ser observado na FIG.5.8.

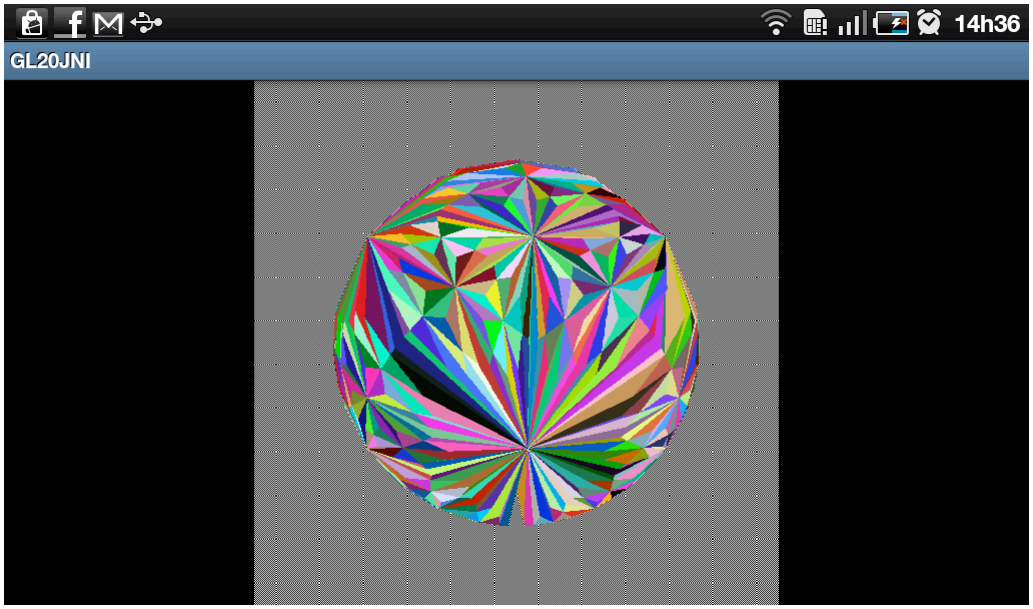

FIG.5.8 *Background* com textura de tamanho máximo.

Também de forma independente do tamanho da textura, percebeu-se que o maior número de faces suportado pelo sólido foi de $4x3^5$ faces, o que corresponde ao nível cinco, como também pode ser observado na FIG.5.8.

6 TRABALHOS FUTUROS

Uma das formas mais eficientes de se desenvolver aplicativos de realidade aumentada é através da utilização de recursos de visão computacional. Para tal, é necessário acessar a câmera do dispositivo. Se este acesso for feito através do código em Java, poderá haver atraso na execução do programa. Dessa forma, deseja-se realizar o acesso utilizando código nativo.

Para efetuar o acesso nativo à câmera do dispositivo, serão explorados os recursos da OpenCV, uma biblioteca voltada para o desenvolvimento de aplicativos de visão computacional.

Assim, a próxima etapa do trabalho consiste no estudo da biblioteca OpenCV e na implementação do acesso nativo à câmera do dispositivo.

Posteriormente, deseja-se modificar o aplicativo relatado no capítulo 5 para que, o último nível de detalhamento do plano de fundo corresponda às imagens obtidas através do acesso nativo à câmera do dispositivo.

7. BIBLIOGRAFIA

MUNSHI, A., GINSBURG, D., SHREINER, D. OpenGL ES 2.0 Programming Guide. Addison-Wesley. 2008.

Android Developer, Dev Guide.

Disponível em: http://developer.android.com/guide/topics/fundamentals.html [Acessado pela primeira vez em: 09 Set 2011].