Universidade do Minho

Escola de Engenharia
Licenciatura em Engenharia Informática

Computação Gráfica

Fase 4

GRUPO: 16
Carlos Sá - 59905
Daniel Andrade - 51782
João Lopes - 61077
Ricardo Varzim - 51814

Maio 2015

1. Introdução

O projecto realizado no âmbito da disciplina de Computação Gráfica é dividido em quatro fases. O presente relatório é destinado a explicar o pretendido e o realizado bem como os métodos adoptados pelo grupo para a realização da quarta (ultima) fase do projecto.

Tal como aconteceu nas fases anteriores, os ficheiros .XML vão voltar a sofrer alterações por forma a contemplarem as novas possibilidades de formato. Assim, mais concretamente, vai ser possível inserir luzes no sistema, através da sua declaração no ficheiro .XML, e de cores e texturas para os vários sólidos. Para a definição das luzes e das cores é possível definir, para além da posição das mesmas, as suas componentes de luz/cor.

Para que tal seja possível foi necessário nesta fase revisitar a aplicação Gerador que gera os pontos das primitivas gráficas. Agora, para além dos pontos, o Gerador, para cada sólido tem de ter a capacidade de gerar as normais, que são utilizadas para iluminar um sólido, bem como as coordenadas de textura para que uma imagem possa ser colocada como textura num sólido.

Desta forma, esta ultima fase do trabalho consiste em duas partes fundamentais: a alteração do Gerador para criar as normais e coordenadas de textura e a alteração do Motor para que seja capaz de ler os novos dados do .XML e aplicar as luzes, cores e texturas.

No método de desenvolvimento adotado para suportar as novas funcionalidades, começou-se por pensar no modo como o novo formato de .XML ia ser lido, a partir das funções disponíveis na biblioteca de *tinyxml*, como gravar a informação lida por forma a poder ser aplicada pelas primitivas disponíveis do *OpenGL* para aplicação de luzes, cores e texturas.

De notar que todo o tipo de transformações (rotações, translações, escalas) implementadas nas fases anteriores se mantém nesta fase.

Um dos objectivos é ainda, tal como na fase anterior, a criação de um ficheiro .XML que seja capaz de conter a informação necessária à implementação de um modelo do sistema solar, sendo este modelo dinâmico, mas agora com a aplicação de texturas para os planetas bem como uma luz (Sol) que ilumina a cena.

Nos capítulos seguintes especificasse com maior detalhe as formas e métodos adoptados para o desenvolvimento dos objectivos pretendidos bem como as funcionalidades adicionais que o grupo considerou relevantes implementar.

2. Normais e Coordenadas de Textura

Uma vez que nesta fase é necessário a criação de normais bem como de coordenadas de textura, então, foi necessário alterar a estrutura de dados do Gerador até agora implementada.

2.1. Estrutura de Dados (Gerador)

A estrutura de dados do Gerador é agora:

```
typedef struct Triangulo {
	//Pontos
	float ponto1[3];
	float ponto2[3];
	float ponto3[3];

	//Normais
	float normal1[3];
	float normal2[3];
	float normal3[3];

	//Texturas
	float text1[2];
	float text2[2];
	float text3[2];
} TRIANGULO;
```

Assim, continua a ter-se a estrutura TRIANGULO já apresentada no relatório da fase 1 do trabalho, contudo, agora tem em consideração a existência de normais e coordenadas de textura.

Como um triângulo possui três vértices, então, tal como para os pontos, vai existir uma normal e uma coordenada de textura para cada vértice. Tal como os pontos, uma normal é constituída por três coordenadas (x, y, z) e as texturas possuem apenas duas coordenadas (x, y).

2.2. Calculo das Normais e Coordenadas de Textura

Tal como para os pontos, as normais e as coordenadas de textura são calculados e armazenados no array de TRIANGULO que existe em cada função de criação de cada sólido. Após o preenchimento desse array com todas as coordenadas de pontos, normais e textura é aplicada a função que grava todas as coordenadas no ficheiro .3d que já foi apresentada no relatório da fase 1 e que apenas foi reformulada para gravar as normais e coordenas de textura.

2.2.1. Esfera

O calculo das normais da esfera segue uma lógica parecida à criação dos pontos. Como foi explícito na fase 1 deste trabalho, a localização dos pontos derivada de dois ângulos (beta, alpha) que são calculados consoante a camada e a fatia atual do ponto, a este calculo é multiplicado o valor do raio da esfera para obtermos a respectiva posição do ponto.

Tendo em conta os cálculos anteriores, bastou retirar a multiplicação do raio para que o resultado gerado seja o valor da normal do ponto em relação à superfície da esfera.

De salientar que esta implementação só é aplicavél à esfera, isto porque os cálculos são efetuados relativamente ao centro da mesma e tratasse de uma superfície de certa forma uniforme para todos os pontos. Caso contrário o calculo da normal teria de ser efetuado tendo em conta todas as normais dos triângulos vizinhos.

O calculo das coordenadas de textura da esfera teve de ser adaptado a cada fase de desenho das camadas. De uma forma geral, e a título de exemplo a formula de calculo para as cordenadas (x,y) foi obtida através do seguinte código:

```
triangulos[nTriangulos].text1[0] = (fatia ) / (float)nfatias;
triangulos[nTriangulos].text1[1] = 1 - ((camada - 1) / (float)ncamadas);
```

Este código varia em função do vértice que representa, sendo que as coordenadas da textura são calculadas para cada vértice dos triângulos, sendo que X corresponde à posição 0 e Y à posição 1 no array Text*i*[*n*] *(i=0,1,2* e *n=0,1)*, desta forma temos a coordenada X calculada com base na fatia a que o vértice corresponde e a coordenada Y com base na camada em que o vértice se encontra. Devido à ordem de desenho da nossa esfera foi ainda necessário inverter a coordenada Y da textura para termos a imagem na posição certa em relação ao topo e fundo da esfera

De notar que as coordenadas são normalizadas tratando assim a imagem da textura com X e Y entre 0 e 1.

2.2.2. Cone

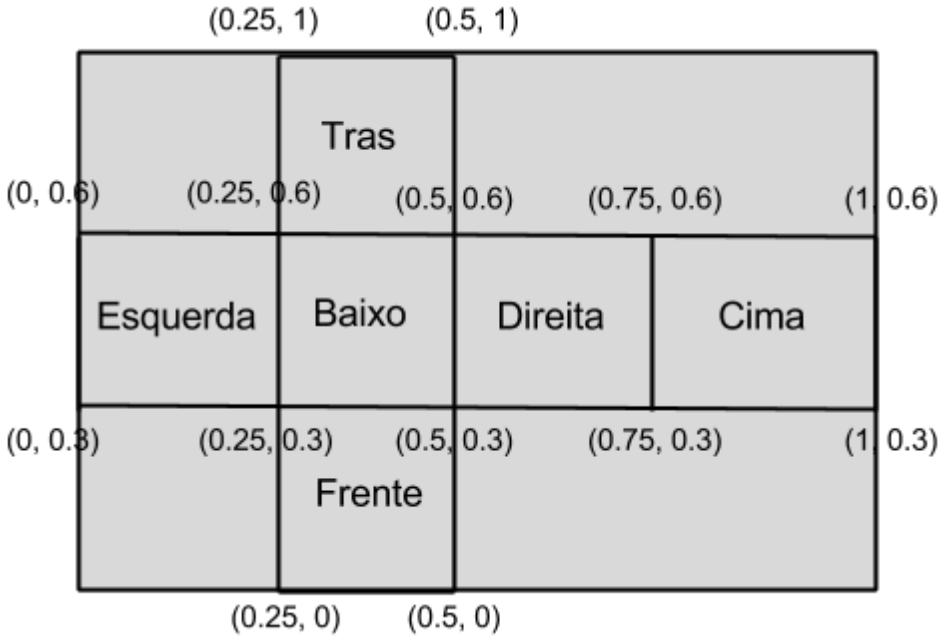
Para esta fase do trabalho, uma vez que o principal objectivo é a aplicação de luzes, cores e texturas ao sistema solar, as funções do gerador relativas ao cone não sofreram alterações, não tendo sido calculadas normais nem coordenadas de textura para o mesmo.

2.2.3. Paralelepipedo

A criação de normais para o paralelepipedo foi efetuada manualmente uma vez que este possui coordenadas que são “diretas”. Mais concretamente, para a face da *frente* de qualquer paralelepipedo, as suas normais vão ser sempre, independentemente do vértice, *(0.0, 0.0, 1.0)*. O mesmo raciocínio aplica-se para as restantes faces do sólido: *cima(0.0,*

1.0, 0.0), *traseira(0.0, 0.0, -1.0)*, *baixo(0.0, -1.0, 0.0)*, *esquerda(-1.0, 0.0, 0.0)* e *direita(1.0, 0.0, 0.0)*.

Para as coordenadas de textura, foram também calculadas manualmente, a foi aplicado o formato habitual de um paralelepipedo (quadrado) cujos valores de coordenadas podem ser observados na figura abaixo:



2.2.4. Plano

Para o plano, tal como para o paralelepipedo, as normais foram calculadas e atribuídas diretamente, uma vez que um plano tem sempre a mesma normal para qualquer vértice: (0.0, 1.0, 0.0).

As coordenas de textura foram também atribuídas manualmente:

3. O Motor

O motor deve agora ter a capacidade de ler um ficheiro .XML e retirar toda a informação até agora considerada e ainda, as luzes, cores, definições de cores e texturas a serem aplicadas.

Desta forma, foi necessário inserir mais informação à estrutura de dados até agora utilizada bem como a criação de novas variáveis globais.

3.1. Estrutura de Dados (Novas e Alterações)

Toda a informação que fazia parte da estrutura de dados “SOLIDO” da fase anterior é mantida nesta fase. Assim, apenas foi acrescentada mais informação que, para melhorar a leitura, apresenta-se apenas esta nova informação inserida na estrutura (toda a informação da fase anterior continua lá):

```
typedef struct Solido {
	float *normalB = (float *)malloc(sizeof(float) * 100000);
	int nNormal = 0; //Contador onde o array de normais vai

	float *texturaB = (float *)malloc(sizeof(float) * 100000);
	int nTextura = 0; //Contador onde o array de coordenadas de textura vai
	int flagTextura = 0; //Flag que permite saber se o solido tem ou nao uma textura
a ser aplicada. 0 -> nao, 1 -> sim!!!
	char nomeTextura[50]; //Nome da textura a aplicar ao solido

	//Materiais para o solido (cor/luz)
	float corSolido[3];
	GLfloat difusa[4];
	GLfloat especular[4];
	GLfloat ambiente[4];
	GLfloat emissiva[4];
	GLfloat shininess;

	//Flags de controlo para saber que tipo de materiais aplicar ao solido
	int flagCorSolido = 0;
	int flagDifusa = 0;
	int flagEspecular = 0;
	int flagAmbiente = 0;
	int flagEmissiva = 0;
	int flagShininess = 0;
} SOLIDO;
```

De salientar novamente que a informação supra apresentada é apenas o que foi acrescentado nesta fase à estrutura “SOLIDO”, a restante informação inserida na fase anterior foi apagada para uma melhor leitura do relatório.

Como é possível observar, existem mais dois arrays de float (“normalB” e “texturaB”) que vão conter os valores/coordenadas das normais e de textura de um sólido para que seja possível aplicar VBO’s.

Um sólido pode agora possuir uma cor e ainda a informação referente às componentes de cor para o mesmo. Assim, foi criada a variável “corSolido[3]” que contém três posições que são os valores RGB lidos do ficheiro .XML. As variáveis: “difusa[4]” a “emissiva[4]” representão as componentes de cor do sólido, com quatro posições, para os valores de RGBA. A ultima é também uma componente de cor do sólido com valor único.

É possível ainda observar um conjunto de flags de controlo do tipo inteiro cujo propósito é saber quais valores de componentes e de cor são realmente para aplicar no sólido uma vez que no ficheiro .XML não é necessário especificar todas as componentes nem uma cor obrigatoriamente. Á frente no relatório vai ser dada mais informação acerca deste processo.

3.1.1. Luzes

Foi considerado pelo grupo que as luzes seriam colocadas sempre no contexto mundo. Mais especificamente, quando uma luz é lida do ficheiro .XML esta vai ser aplicada ao mundo na posição especificada e não a um objecto (ou sólido) ou seja, as luzes não vão sofrer transformações como translações ou rotações.

Assim, tendo em conta o contexto mundo, foi criada mais uma estrutura de dados:

```
typedef struct Luz {
	GLfloat posicao[4];
	GLfloat ambiente[4];
	GLfloat difusa[4];
	GLfloat especular[4];
	GLfloat spotDir[3];
	GLfloat spotCutof;
	GLfloat spotExponent;
} LUZES;
```

Esta estrutura “funciona em paralelo” com a variável global:

```
LUZES luzes[20]; //Limitacao de ate 20 luzes
```

É portanto declarado um array (limitado a 20 posições/luzes) de “LUZES”. Uma luz é constituída pela sua posição que contém quatro posições, as três primeiras para as coordenadas X,Y,Z onde vai ser colocada e a ultima posição para determinar o tipo de luz a implementar (direcional, ponto ou spotlight). Os restantes atributos das luzes são os que o grupo considerou implementar e a sua descrição vai em conformidade com o que é requerido pelo OpenGL.

Sempre que na leitura do ficheiro .XML seja encontrada uma luz, então é feita uma nova entrada no array global de luzes e os valores dos atributos das luzes são inicializados para as pré-definições (defaults) antes de ser lido os valores dos atributos especificados no .XML.

A função que inicializa os valores dos atributos das luzes é:

```
void initLuzes(int posicao) {
	luzes[posicao].posicao[0] = (float)0;
	luzes[posicao].posicao[1] = (float)0;
	luzes[posicao].posicao[2] = (float)0;
	luzes[posicao].posicao[3] = (float)0;
	luzes[posicao].ambiente[0] = (float)0;
	luzes[posicao].ambiente[1] = (float)0;
	luzes[posicao].ambiente[2] = (float)0;
	luzes[posicao].ambiente[3] = (float)0;
	luzes[posicao].difusa[0] = (float)1;
	luzes[posicao].difusa[1] = (float)1;
	luzes[posicao].difusa[2] = (float)1;
	luzes[posicao].difusa[3] = (float)1;
	luzes[posicao].especular[0] = (float)0.75;
	luzes[posicao].especular[1] = (float)0.75;
	luzes[posicao].especular[2] = (float)0.75;
	luzes[posicao].especular[3] = (float)1;
	luzes[posicao].spotDir[0] = (float)0;
	luzes[posicao].spotDir[1] = (float)0;
	luzes[posicao].spotDir[2] = (float)1;
	luzes[posicao].spotCutof = (float)180;
	luzes[posicao].spotExponent = (float)0;
}
```

Os valores considerados default foram consultados na internet cujos links se encontram na bibliografia. (Os valores para a componente especular não são os considerados default mas são valores inseridos pelo grupo para testes). À frente, no relatório, quando for explicado a forma de aplicar as luzes é explicado com mais precisão o porquê da implementação/inicialização destes valores.

3.2. VBOs (Vertex Buffer Objects)

Uma vez que na fase 3 foi necessário substituir o modo direto de desenho das figuras passando a ser desenhadas tirando partido de VBOs levou à necessidade de nesta fase continuar com a aplicação de VBOs. Assim, como referido acima, cada solido possui um array de normais e outro array de coordenadas de textura que vão ser utilizados pelas funções de VBOs, em conjunto com o array de pontos que já vem da fase anterior, para desenhar o sólido com respetivas normais (cor/luz) e textura.

Tal como na fase anterior, à medida que as normais e as coordenadas de textura de um sólido são lidos, vão sendo escritos na mesma ordem para o respetivo array. Após

todos os sólidos, normais e coordenadas de textura a desenhar se encontrarem em memória nos respetivos arrays é então possível criar os VBOs a partir da função:

```
void iniciarVBOs() {
	glEnableClientState(GL_VERTEX_ARRAY);
	glEnableClientState(GL_NORMAL_ARRAY);
	if (nTexturas > 0) glEnableClientState(GL_TEXTURE_COORD_ARRAY);

	buffers = (GLuint *)malloc(sizeof(GLuint) * (nSolidos)); //Aloca memoria para o
array de pontos (mesmo tamanho que o array de solidos quando ja preenchido)
	normais = (GLuint *)malloc(sizeof(GLuint) * (nSolidos)); //Aloca memoria para o
array de normais (mesmo tamanho que o array de solidos quando ja preenchido)
	texturas = (GLuint *)malloc(sizeof(GLuint) * (nSolidos)); //Aloca memoria para o
array texturas (mesmo tamanho que o array de solidos quando ja preencgido)

	glGenBuffers(nSolidos, buffers);
	glGenBuffers(nSolidos, normais);
	glGenBuffers(nSolidos, texturas);

	for (int i = 0; i < nSolidos; i++) {
		glBindBuffer(GL_ARRAY_BUFFER, buffers[i]);
		glBufferData(GL_ARRAY_BUFFER, sizeof(float) * solidos[i].nVertex,
solidos[i].vertexB, GL_STATIC_DRAW);

		glBindBuffer(GL_ARRAY_BUFFER, normais[i]);
		glBufferData(GL_ARRAY_BUFFER, sizeof(float) * solidos[i].nNormal,
solidos[i].normalB, GL_STATIC_DRAW);

		glBindBuffer(GL_ARRAY_BUFFER, texturas[i]);
		glBufferData(GL_ARRAY_BUFFER, sizeof(float) * solidos[i].nTextura,
solidos[i].texturaB, GL_STATIC_DRAW);
	}
}
```

A partir desta função e tirando partido das variáveis globais “buffers”, “normais” e “texturas” é possível desenhar as figuras geométricas com respetivas normais e texturas da forma habitual quando se utiliza VBOs.

3.3. Colocar Materiais

Antes de proceder ao desenho de um determinado sólido, é primeiro inserido os seus materiais.

A função de inserção/implementação dos materiais faz uso das seguintes variáveis globais:

```
//Variaveis globais com os valores considerados default para as cores dos solidos
GLfloat defaultDifusa[4] = {0.8, 0.8, 0.8, 1.0};
GLfloat defaultAmbiente[4] = {0.2, 0.2, 0.2, 1.0};
GLfloat defaultEspecular[4] = {0.75, 0.75, 0.75, 1.0};
GLfloat defaultEmissiva[4] = {0.0, 0.0, 0.0, 1.0};
GLfloat defaultShininess = 128.0;
```

Estas variáveis globais representam os valores por default para as várias componentes de cor. Tais valores foram consultados na internet cujos links se encontram na bibliografia. Alguns dos valores apresentados não são mesmo os valores de default porque foram alterados pelo grupo durante testes.

A função que aplica os materiais a um determinado sólido é:

```
void colocaMaterialSolido(int posSolido) {
	//printf("CORES
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
	//printf("Solido: %d Flag: %d NumSolidos: %d\n", posSolido,
solidos[posSolido].flagCorSolido, nSolidos);
	if (solidos[posSolido].flagCorSolido == 1)
glColor3f(solidos[posSolido].corSolido[0], solidos[posSolido].corSolido[1],
solidos[posSolido].corSolido[2]);
	else glColor3f(1.0, 1.0, 1.0);
	if (solidos[posSolido].flagDifusa == 1) glMaterialfv(GL_FRONT, GL_DIFFUSE,
solidos[posSolido].difusa);
	else glMaterialfv(GL_FRONT, GL_DIFFUSE, defaultDifusa);
	if (solidos[posSolido].flagEspecular == 1) glMaterialfv(GL_FRONT, GL_SPECULAR,
solidos[posSolido].especular);
	else glMaterialfv(GL_FRONT, GL_SPECULAR, defaultEspecular);
	if (solidos[posSolido].flagAmbiente == 1) glMaterialfv(GL_FRONT, GL_AMBIENT,
solidos[posSolido].ambiente);
	else glMaterialfv(GL_FRONT, GL_AMBIENT, defaultAmbiente);
	if (solidos[posSolido].flagEmissiva == 1) glMaterialfv(GL_FRONT, GL_EMISSION,
solidos[posSolido].emissiva);
	else glMaterialfv(GL_FRONT, GL_EMISSION, defaultEmissiva);
	if (solidos[posSolido].flagShininess == 1) glMaterialf(GL_FRONT, GL_SHININESS,
solidos[posSolido].shininess);
	else {
		glMaterialf(GL_FRONT, GL_SHININESS, defaultShininess);
		//printf("SHININESS: %f\n", defaultShininess);
	}
}
```

Os materiais são aplicados ao sólido que se encontra na posição, do array de sólidos, passada como argumento à função.

Para cada componente de cor, a função vai verificar a sua flag, para o sólido em questão, para determinar se essa componente foi especificada no ficheiro .XML. Caso tenha sido, então são aplicados os valores que foram lidos do .XML e guardados em memória nas componentes, caso contrário é aplicado os valores considerados default.

Desta forma, todas as componentes de cor são sempre aplicadas para cada sólido, contudo, caso não seja especificado qualquer informação acerca dos materiais no ficheiro .XML então são aplicados os valores default que não apresentam qualquer mudança em comparação à não aplicação de materiais. Caso seja especificado qualquer componente de cor no .XML então é aplicada com os valores lidos e as restantes componentes com os valores default que não vão interferir no resultado final do material.

3.4. Colocar Luzes

Uma vez que foi considerado que as luzes existem no contexto mundo como explicado acima no relatório, então estas são colocadas sempre antes de qualquer transformação (translações/rotações).

A função responsável por colocar as várias luzes no mundo é:

```
void colocaLuzes() {
	for (int i = 0; i < nLuzes; i++) {
		glLightfv(GL_LIGHT0 + i, GL_POSITION, luzes[i].posicao); //Posicao
		glLightfv(GL_LIGHT0 + i, GL_AMBIENT, luzes[i].ambiente); //Cor
		glLightfv(GL_LIGHT0 + i, GL_DIFFUSE, luzes[i].difusa); //Cor
		glLightfv(GL_LIGHT0 + i, GL_SPECULAR, luzes[i].especular); //Cor
		if (luzes[i].posicao[3] == (float)-1) { //Caso seja uma spotlight tem de
se passar mais valores para serem desenhados !!!!!!!!!!!!!!!!!!
			glLightfv(GL_LIGHT0 + i, GL_SPOT_DIRECTION, luzes[i].spotDir);
			glLightf(GL_LIGHT0 + i, GL_SPOT_CUTOFF, luzes[i].spotCutof);
			glLightf(GL_LIGHT0 + i, GL_SPOT_EXPONENT, luzes[i].spotExponent);
		}
	}
}
```

Assim, quando esta função é chamada é percorrido o array global de luzes, que foi apresentado acima no relatório, e para cada luz nele presente são implementadas as suas propriedades (posição, ambiente, difusa, especular e caso se trate de uma spotlight são aplicados as restantes propriedades que necessita).

Desta forma, tal como na colocação de materiais, todas as propriedades da luz são desenhadas independentemente se foram todas especificadas no ficheiro .XML. Como quando é encontrada uma luz nova no .XML todas as suas propriedades são iniciadas com os valores considerados default, então, estes não vão influenciar o resultado final da luz para as propriedades especificadas da mesma. Tem um comportamento similar aos materiais.

As luzes são ligadas na *main()* com o código:

```
//Ligar as luzes
if (flagLuz == 1) {
	//Ligar o quadro:
	glEnable(GL_LIGHTING);
	//Ligar as varias luzes:
	for (int i = 0; i < nLuzes; i++) {
		glEnable(GL_LIGHT0 + i);
	}
}
```

A variável “flagLuz” é uma variável global que se encontra inicialmente a 0 (zero) e passa a 1 (um) mal seja encontrada uma luz no ficheiro .XML a ser aplicada. Serve portanto como um controlo para saber se existem luzes a ser aplicadas ou não.

A função começa por “ligar o quadro” e depois liga cada uma das luzes individualmente até ao numero de luzes total que existe.

3.5. Texturas

Como é possível observar na estrutura de dados de “SOLIDO” apresentada acima, um sólido possui uma “flagTextura” e “nomeTextura[50]”. A flag é iniciada a 0 (zero) e é utilizada para saber se um determinado sólido vai possuir uma textura a ser desenhada ou não. Na leitura do ficheiro .XML, quando é encontrada uma textura para um determinado sólido, a flagTextura passa a 1 (um) e é guardado o nome da textura na variável “nomeTextura” para que seja utilizada pelas funções de DevIL ao ser desenhada.

Após todas as texturas a desenhar de cada sólido estejam em memória com respetivas coordenadas e VBOs iniciados é então chamada a função que “prepara” as texturas a desenhar:

```
void preparaTexturas() {
	unsigned int ima, imaW, imaH;
	unsigned char *imaData;
	int posicaoTextura = 0; //Variavel para saber a posicao de textura a aplicar

	texturasID = (GLuint *)malloc(sizeof(GLuint) * (nSolidos)); //Aloca memoria para
o array de ID de texturas (no maximo tem o mesmo tamanho que o numero de solidos)

	glGenTextures(nTexturas, texturasID);

	//Percorre todos os solidos para verificar quais possuem texturas a desenhar
	for (int i = 0; i < nSolidos; i++) {
		if (solidos[i].flagTextura == 1) {
			ilGenImages(1, &ima);
			ilBindImage(ima);
			ilLoadImage((ILstring) solidos[i].nomeTextura);
			imaW = ilGetInteger(IL_IMAGE_WIDTH);
			imaH = ilGetInteger(IL_IMAGE_HEIGHT);
			ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
			imaData = ilGetData();

			glBindTexture(GL_TEXTURE_2D, texturasID[posicaoTextura]);
			glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
			glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
			glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
			glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
			glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, imaW, imaH, 0, GL_RGBA,
GL_UNSIGNED_BYTE, imaData);

			posicaoTextura++; //Atualiza o contador para "apontar" para a
proxima textura a ser desenhada
		}
	}
}
```

A função toma partido de várias variáveis tanto locais (da própria função) como globais que foram todas criadas devido à necessidade de serem utilizadas pelas funções habituais de DevIL.

Para que fosse possível ter sólidos com textura e sem textura na mesma "cena" foi necessário criar uma variável que serve como contador do array "texturasID" em vez de ser utilizada a posição do sólido em cada momento, sendo essa variável "posicaoTextura". Pode-se desta forma ter vários sólidos sem textura e outros com textura, independentemente da ordem.

À parte da criação da variável acima exposta, esta função segue exatamente a mesma estrutura disponibilizada nos slides das aulas pelo docente da disciplina.

3.6. Desenho das Figuras & Aplicação das Transformações

Para esta fase do trabalho, tudo o que diz respeito a transformações (translações, rotações, escalas) não sofreu qualquer alteração em comparação com a fase anterior. Foi apenas acrescentada à função responsável pela aplicação das transformações e desenho dos sólidos a aplicação das luzes no mundo e dos materiais e texturas em cada sólido.

Apresenta-se essa função, que para uma melhor leitura foram apagadas as partes que correspondem às transformações geométricas:

```
void desenharSolidos() {
	float res[3];
	int auxText = 0;

	glEnable(GL_NORMALIZE);
	if (flagLuz == 1) colocaLuzes();

	for (int i = 0; i < nSolidos; i++) { //Percorre o array de solidos
		glPushMatrix();

		//PARTE DAS TRANSFORMAÇÕES GEOMÉTRICAS AQUI

		//Desenha o solido que esta no indice (i) actual
		colocaMaterialSolido(i);
		glBindBuffer(GL_ARRAY_BUFFER, buffers[i]);
		glVertexPointer(3, GL_FLOAT, 0, 0);
		glBindBuffer(GL_ARRAY_BUFFER, normais[i]);
		glNormalPointer(GL_FLOAT, 0, 0);
		//Aplica texturas caso tenha textura para ser aplicada
		if (solidos[i].flagTextura == 1) {
			glBindBuffer(GL_ARRAY_BUFFER, texturas[i]);
			glTexCoordPointer(2, GL_FLOAT, 0, 0);
			glBindTexture(GL_TEXTURE_2D, texturasID[auxText]);
			auxText++;
		}
		glDrawArrays(GL_TRIANGLES, 0, solidos[i].totalVertices);
		if (solidos[i].flagTextura == 1) glBindTexture(GL_TEXTURE_2D, 0);
```

```
                //Faz o pop matrix
                glPopMatrix();
        }
}
```

É possível observar que logo no inicio da função são aplicadas as luzes caso existam através da função “colocaLuzes()” já apresentada.

Na parte de código responsável por desenhar os sólidos, a primeira operação agora realizada é a aplicação dos materiais no sólido através da função “colocaMaterialSolido(posSolido)” também já apresentada.

São depois processados os pontos e as normais do sólido e por fim é testado se este possui uma textura a ser aplicada. Caso possua é então aplicado o código responsável pelo desenho das mesma (com a mesma estrutura dos slides das aulas). É depois desenhado o sólido e por fim caso o mesmo possua uma textura é retomado a 0 (zero) na função “glBindTexture”.

3.7. XML

Apresenta-se uma possível estrutura de um ficheiro .XML para esta fase:

```
<cena>
        <luzes>
                <luz tipo="POINT" posX=-4 posY=4 posZ=4 />
        </luzes>
        <grupo>
                <modelos>
                        <modelo ficheiro="f4_plano1.3d" textura="planotextura1.jpg"
diffR=1.0 diffG=1.0 diffB=1.0 emiR=0.8 emiG=0.8 emiB=0.5 />
                </modelos>
                <grupo>
                        <translacao X=1 Y=1 Z=1 />
                        <modelos>
                                <modelo ficheiro="f4_plano1.3d" diffR=1.0 diffG=1.0
diffB=0.7 />
                        </modelos>
                        <grupo>
                                <translacao X=1 Y=1 Z=1 />
                                <modelos>
                                        <modelo ficheiro="f4_plano1.3d"
textura="planotextura1.jpg" diffR=1.0 diffG=1.0 diffB=0.7 />
                                </modelos>
                        </grupo>
                </grupo>
        </grupo>
        <grupo>
                <translacao X=0 Y=2 Z=0 />
                <rotacao tempo=5 eixoX=0 eixoY=1 eixoZ=0 />
                <modelos>
                        <modelo ficheiro="f4_plano1.3d" textura="text_plano.jpg" emiR=0.6
emiG=0.6 emiB=0.6 />
```

```
            </modelos>
        </grupo>
</cena>
```

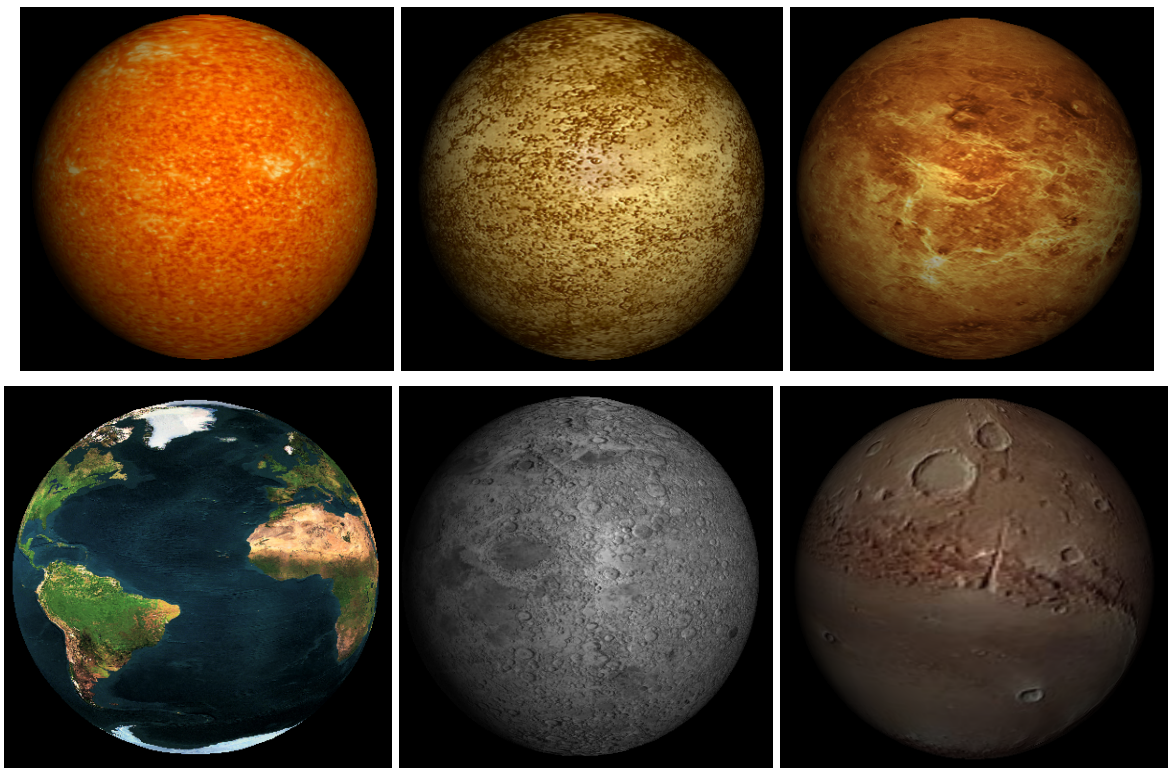
É possível observar a existência de uma luz do tipo “POINT” e especificação da sua localização. O exemplo possui ainda modelos (sólidos) a desenhar com e sem textura neles e definições das componentes de cor.

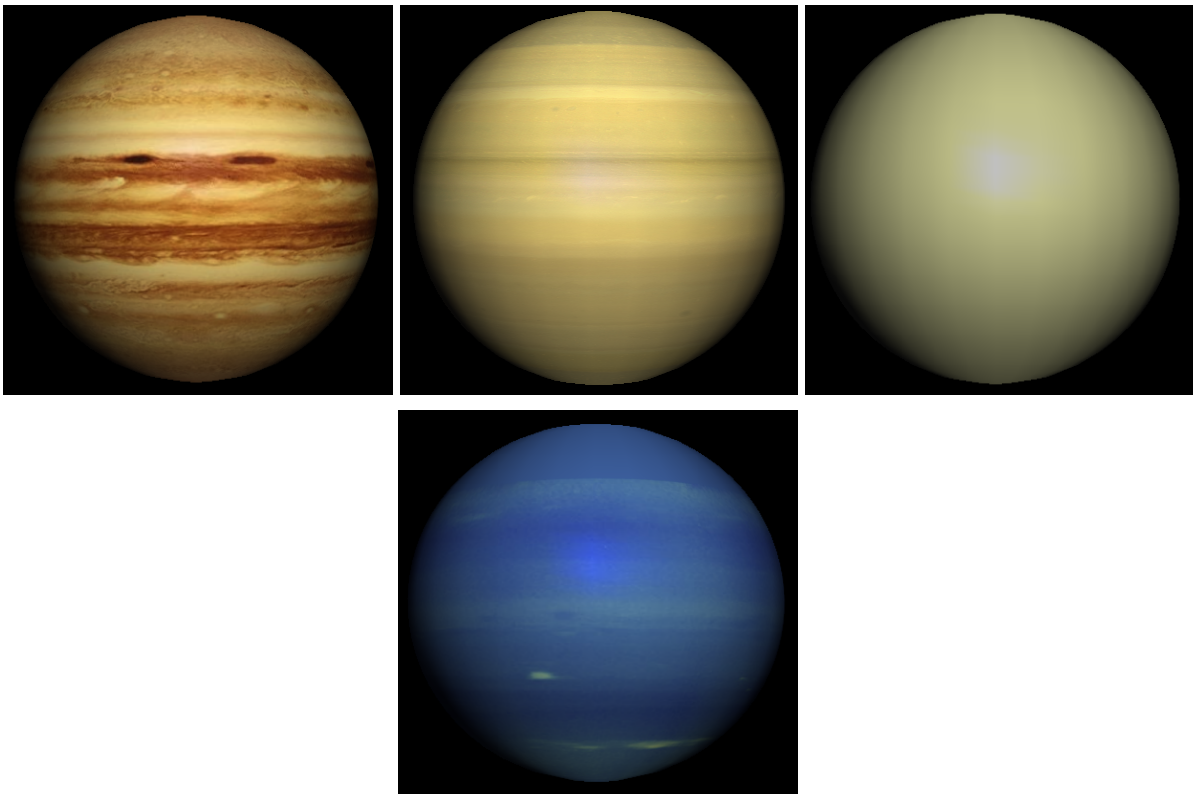
Para ler o ficheiro .XML tal como nas fases anteriores foi utilizada a biblioteca *tinyxml*.

As funções XML implementadas permitem retirar toda a informação relevante nos ficheiros. A leitura do XML e povoamento das estruturas de dados tem o mesmo comportamento como nas fases anteriores, sendo apenas adaptado para extrair nova informação e povoar novas variáveis.

3.8. Planetas

Abaixo é possível observar o Sol e cada um dos planetas e a nossa Lua, com luz aplicada. As imagens foram obtidas a partir de captura de ecrã de vários modelos .XML criados para cada planeta, Sol e Lua, que também se encontram na pasta submetida com o trabalho.





(Sol, Mercúrio, Vénus, Terra, Lua, Marte, Júpiter, Saturno, Urano, Neptuno)

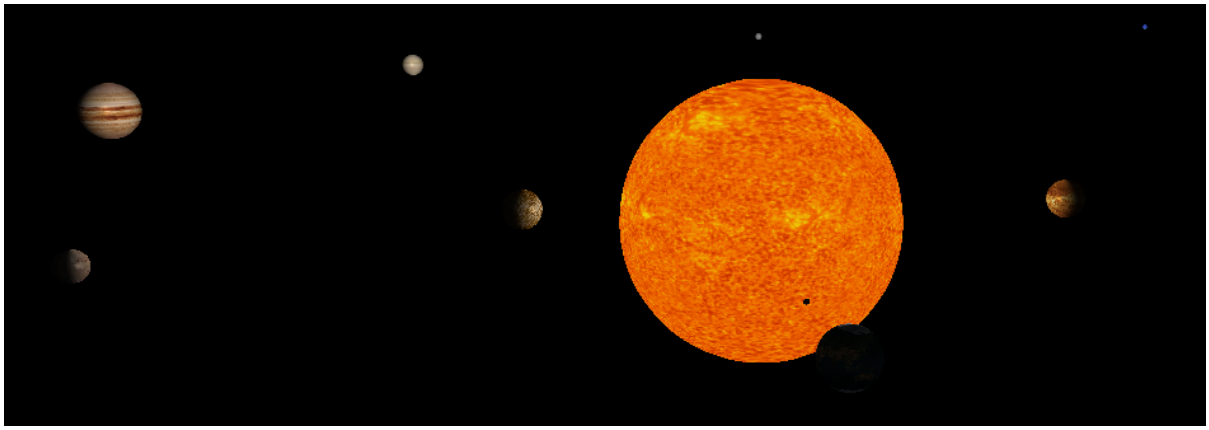
3.9. Modelo do Sistema Solar (Dinâmico) com Luzes, Materiais e Texturas

Um dos objectivos para esta ultima fase do trabalho era a implementação de uma luz para o Sol e de aplicação de texturas para os planetas no modelo de sistema solar dinâmico já implementado na fase anteriror.

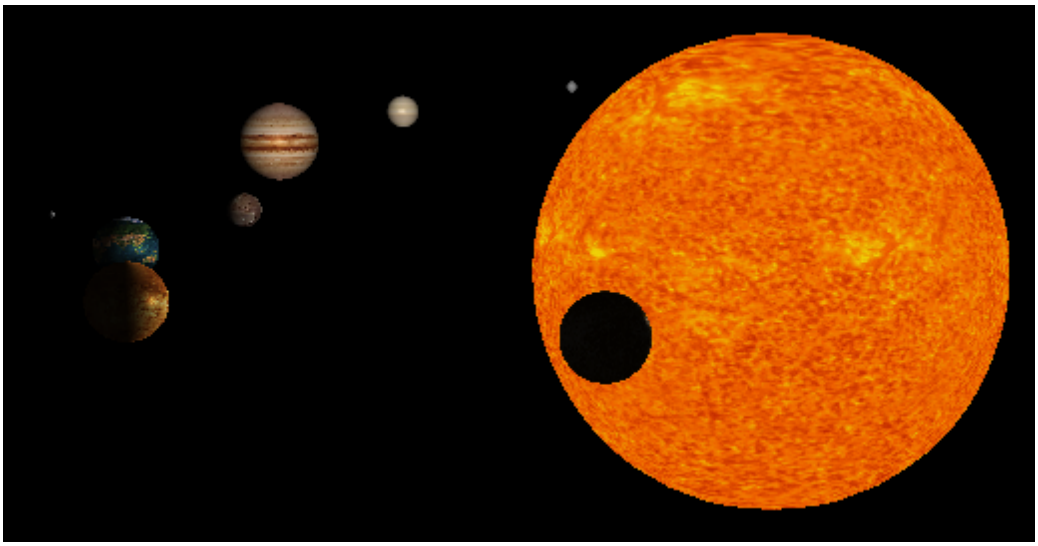
Tal como na fase anterior, o grupo decidiu fazer o mesmo de forma mais realista possível, sem comprometer a visualização do mesmo. Ao contrário da fase 2, onde todas as distâncias e escalas representavam o sistema solar à escala, o que levou a ter um Sol muito grande comparado com os planetas que ficavam demasiado pequenos, agora apenas é utilizada a translação como medida realista, exatamente do mesmo modo como na fase anterior.

No modelo criado, pode ser observado o Sol, os oito planetas e a nossa Lua, todos eles com movimentos de translação e de rotação sobre eles próprios. No centro do Sol foi colocada uma luz (a única luz do modelo) do tipo “point” e são aplicadas texturas realistas a cada planeta cujos links se encontram na bibliografia. A iluminação do próprio Sol é efetuada declarando no ficheiro .XML que este possui uma cor emissiva (em tons de laranja). Tal como na fase anterior, os planetas estão referentes ao Sol como seria de esperar.

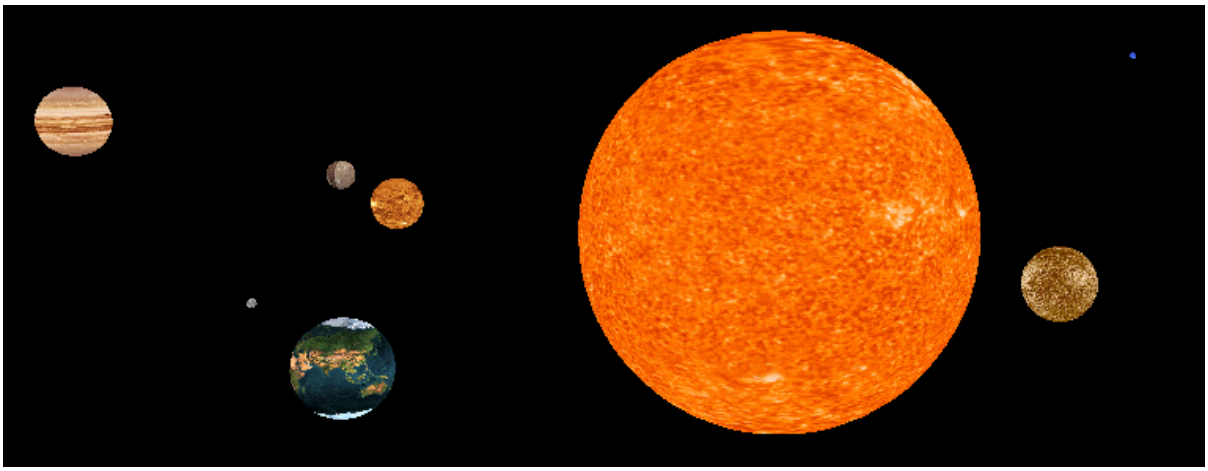
O ficheiro .XML que contém o sistema solar tem nome: "`xml_f4_sistsolar`".



Sol e os oito planetas (com a Lua) com luz ligada



Sol e alguns dos planetas com luz ligada



Sol e alguns dos planetas sem luz ligada

3.10. Ficheiros

Como pedido, junto com as pastas do projecto, existem duas pastas específicas para cada um dos dois tipos de ficheiros (.3d e .XML) de nomes “Ficheiros 3d” e “Ficheiros XML”. Dentro da pasta de ficheiros .XML existem quatro pastas, cada uma com os ficheiros de cada fase. Os ficheiros contidos na pasta desta fase encontram-se também na pasta do Motor para que estejam prontos a serem executados/lidos.

Dentro da pasta de ficheiros .3d existem duas pastas, uma com os ficheiros gerados e utilizados para as primeiras três fases e outra pasta com os ficheiros .3d gerados para esta fase do trabalho uma vez que agora existem normais e coordenadas de textura.

Os ficheiros .XML desta segunda fase têm nomes de forma: “`xml_f4_exi`” onde i = 1 até N à excessão do ficheiro do modelo do sistema solar cujo nome está no capítulo acima e dos ficheiros de cada planeta que tem a forma: “`xml_f4_nomeplaneta`”.

3.11. Menus

Tal como na fase anterior, o menu até agora implementado é acessível a partir de “click down” na “mouse-weel”. Nesta fase todas as opções no menu até agora disponíveis continuam disponíveis e foi acrescentada a possibilidade de ligar e desligar as luzes. Esta nova funcionalidade/opção tornou-se bastante útil para comparar de forma imediata resultados com e sem luz por forma a perceber se a luz tem um comportamento adequado/correto.

4. Apreciação Crítica & Conclusão

Com a resolução deste trabalho o grupo foi capaz de adquirir novos conhecimentos, na linguagem C que era já conhecida por todos os elementos do grupo mas mais concretamente em XML e como utilizar a biblioteca de *tinyxml* que permitiu ler os vários ficheiros criados com cenas a desenhar e ainda conhecimentos sobre computação gráfica, mais concretamente, na utilização de OpenGL.

Para todos os elementos do grupo este trabalho foi a primeira vez que tivemos contacto direto no desenvolvimento de primitivas gráficas e aplicação de métodos de desenho pelo que o grupo tem noção que a quantidade de conhecimento adquirido é apenas um conhecimento inicial/geral acerca desta área contudo consideramos que o problema proposto foi suficientemente geral por forma a por em prática todos os conhecimentos adquiridos e ter contacto com limitações e problemas que podem ocorrer em determinadas situações.

Na terceira fase do trabalho era pedido que fosse criado um bule de chá à custa de patches de Bezier o qual não tivemos tempo de realizar na fase correspondente. O grupo planeou a sua elaboração para esta ultima fase contudo, devido ao numero de trabalhos de todas as disciplinas e ao facto de esta fase, embora simples, mostrou-se em certas ocasiões um desafio uma vez que basta haver erros no calculo das normais e/ou coordenadas de textura para que o resultado final fique comprometido, não foi possível construir o bule de chá.

À parte da construção do bule de chá, para cada uma das fases do trabalho, o grupo acredita ter conseguido atingir todos os objectivos propostos e por diversas vezes foram construídos alguns extras como extensões ao XML e desenho de outras formas (como o boneco de neve da fase 2). Em cada uma das fases ocorreu sempre algum pormenor mais desafiante que acabou por ser ultrapassado.

Como trabalho futuro/continuidade ao trabalho já efetuado, o grupo gostava de ter implementado uma câmara do modo FPS para que fosse possível observar de forma mais direta cada um dos planetas e ainda uma SkyBox para que o plano de fundo fosse constituído por um padrão estrelar por forma a aumentar o realismo da cena. A técnica de picking por forma a identificar o Sol e cada um dos planetas quando clicados e ainda aparecer uma caixa com texto de informação acerca do conteúdo clicado foi também uma das ideias que o grupo gostaria de ter implementado.

5. Bibliografia

Links gerais:
http://resumbrae.com/ub/dms424_s05/10/print.html
https://www.opengl.org/discussion_boards/showthread.php/132502-Color-tables
http://www.falloutsoftware.com/tutorials/gl/gl8.htm
http://stackoverflow.com/questions/20177604/opengl-light-direction
http://stackoverflow.com/questions/8494942/why-does-my-color-go-away-when-i-enable-lighting-in-opengl
http://stackoverflow.com/questions/10181201/opengl-light-changes-ambient-to-diffuse-or-specular-works-but-not-the-opposite
http://math.hws.edu/graphicsnotes/c4/s2.html

Links para as imagens de textura utilizadas:
http://solarviews.com/raw/sun/suncyl1.jpg
http://www.mapsharing.org/MS-maps/map-pages-planets-map/images-planets-map/7-planet-mercury-map.jpg
http://www.solarsystemscope.com/nexus/content/planet_textures/texture_venus_surface.jpg
http://britton.disted.camosun.bc.ca/polymap/earth-map-huge.jpg
http://maps.jpl.nasa.gov/textures/ear1ccc2.jpg
http://www.astrosurf.com/nunes/render/maps/full/mars.jpg
http://maps.jpl.nasa.gov/textures/jup0vtt2.jpg
http://www.mmedia.is/~bjj/data/saturn/saturn.jpg
http://t1.rbxcdn.com/86c5d99679a1ec57e28be9b98f5dacd5
http://www.planetaryvisions.com/libsamples/37_1.jpg