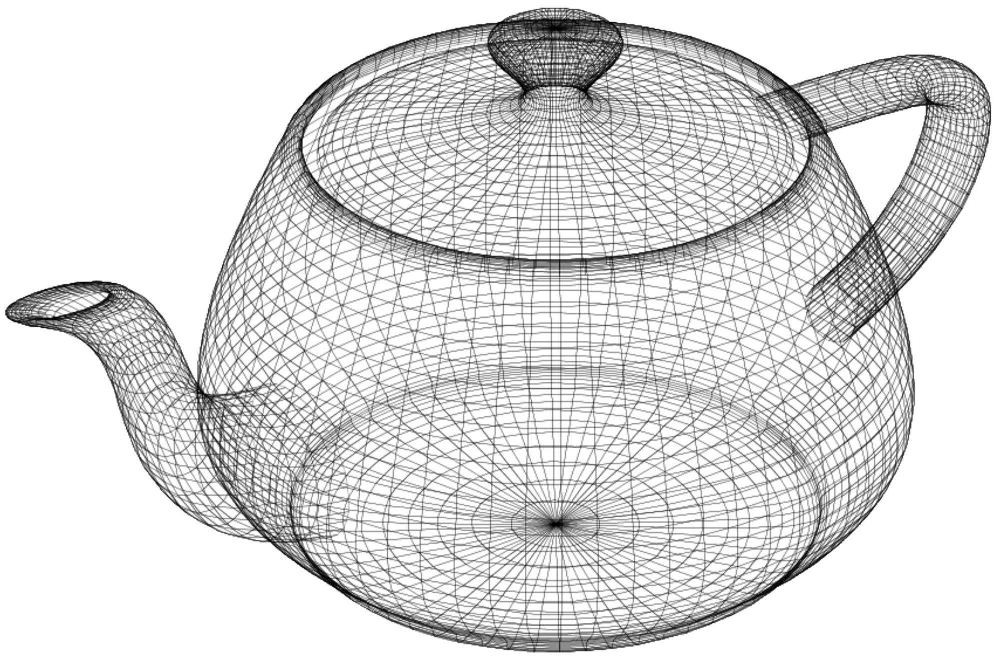
Universidade do Minho
Escola de Engenharia

Computação Gráfica 2024/25: Fase III

CURVAS, SUPERFÍCIES CÚBICAS & VBO'S

Abril 2025



Gustavo Barros
A100656

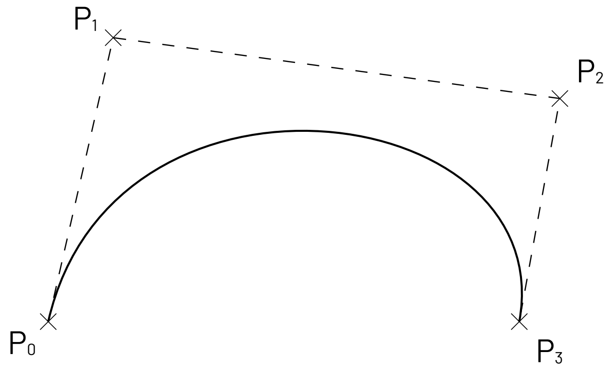
Pedro Ferreira
A97646

Enzo Vieira
A98352

Luís Figueiredo
A100549

1. Curvas Cúbicas de Bézier

Considere-se quatro pontos de controlo $P_{0..3}(x, y, z)$ e que se pretende traçar uma curva entre P_0 e P_3 (designados os "nós" da curva) sob a influência de P_2 e P_3. Adicionalmente, defina-se o progresso na travessia da curva com a variável $t \in [0, 1]$.



A posição na curva para um valor t arbitrário dá-se pela expressão abaixo. Note-se que deve ser calculada para as três componentes do ponto.

$$P(t) = [x(t) \;\; y(t) \;\; z(t)] \tag{1}$$

$$\begin{aligned} P(t) &= (1-t)^3 \cdot P_0 + 3t(1-t) \cdot P_1 + 3t^2(1-t) \cdot P_2 + t^3 \cdot P_3 \\ &= \sum_{i=0}^{3} B_i^3(t) P_i = \sum_{i=0}^{3} \binom{3}{i} \cdot t^i (1-t)^{n-i} \cdot P_i \end{aligned} \tag{2}$$

A reformulação desta expressão em forma matricial resulta numa matriz MP de coeficientes constantes para toda a curva. Portanto, a fim de tornar a avaliação para t mais eficiente, recomenda-se que essa matriz seja calculada *a priori* ao invés de calcular os polinómios de Bernstein diretamente para tantos pontos quanto o nível de tesselação obrigar.

$$\begin{aligned} (1-t)^3 = A(t) &= -t^3 + 3t^2 - 3t + 1 \\ 3t(1-t)^2 = B(t) &= 3t^3 - 6t^2 + 3t \\ 3t^2(1-t) = C(t) &= -3t^3 + 3t^2 \\ t^3 = D(t) &= t^3 \end{aligned} \tag{3}$$

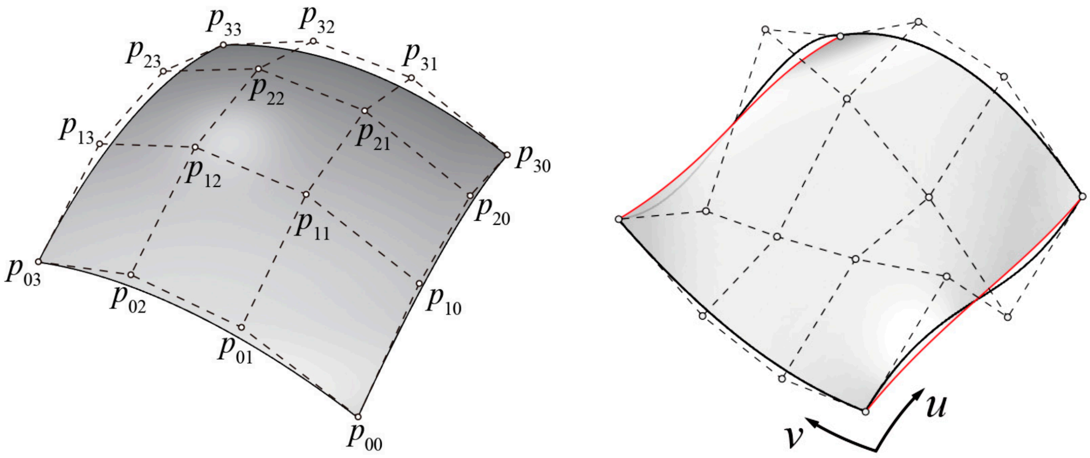
$$P(t) = [A(t) \;\; B(t) \;\; C(t) \;\; D(t)] \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = [t^3 \;\; t^2 \;\; t \;\; 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = [t^3 \;\; t^2 \;\; t \;\; 1] M_{\text{Bézier}} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{4}$$

Para um ponto na curva, a sua derivada pode ser interpretada cineticamente como a sua "direção" atual. Isto é uma noção útil no encadeamento de curvas, pois garante continuidade de 1º grau.

$$P'(t) = [3t^2 \;\; 2t \;\; 1 \;\; 0] M_{\text{Bézier}} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{5}$$

2. Superfícies Bicúbicas de Bézier

O raciocínio das curvas pode ser estendido para uma segunda dimensão - em vez de avaliar a posição num segmento, avalia-se a posição numa superfície. Verifique-se que, se uma curva cúbica possui 4 pontos de controlo, então uma superfície quadrangular com arestas definidas por curvas cúbicas (um *patch* de Bézier) terá $4^2 = 16$ pontos de controlo.



Com duas dimensões é implícita a existência de duas coordenadas: $u, v \in [0, 1]$ (horizontal e vertical).

$$P(u, v) = [x(u, v) \;\; y(u, v) \;\; z(u, v)] \tag{6}$$

$$P(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i^3(u) B_j^3(v) P_{\text{ij}} \tag{7}$$

Para duas dimensões mantém-se também a lógica de haver uma matriz de coeficientes constantes para toda a superfície, neste caso MPM^T:

$$P(u, v) = [u^3 \;\; u^2 \;\; u \;\; 1] M_{\text{Bézier}} \begin{bmatrix} P_{00} & \cdots & \cdots & P_{30} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ P_{03} & \cdots & \cdots & P_{33} \end{bmatrix} M_{\text{Bézier}}^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{8}$$

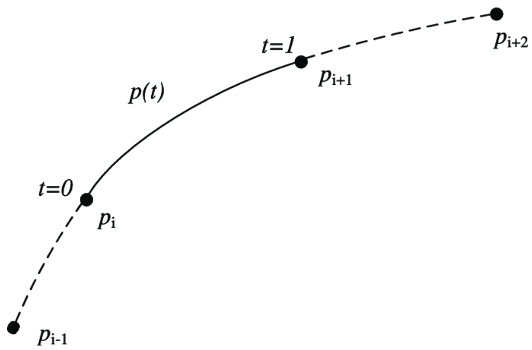
Para um ponto na superfície, as suas derivadas parciais são especialmente úteis para determinar (por produto vetorial normalizado) o seu vetor normal, indispensável à iluminação do modelo.

$$\frac{\delta}{\delta u} P(u, v) = [3u^2 \;\; 2u \;\; 1 \;\; 0] M_{\text{Bézier}} \begin{bmatrix} P_{00} & \cdots & \cdots & P_{30} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ P_{03} & \cdots & \cdots & P_{33} \end{bmatrix} M_{\text{Bézier}}^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{9}$$

$$\frac{\delta}{\delta v} P(u, v) = [u^3 \;\; u^2 \;\; u \;\; 1] M_{\text{Bézier}} \begin{bmatrix} P_{00} & \cdots & \cdots & P_{30} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ P_{03} & \cdots & \cdots & P_{33} \end{bmatrix} M_{\text{Bézier}}^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix} \tag{10}$$

3. Curvas Cúbicas Catmull-Rom

Os nós desta categoria de curvas são P_1 e P_2 - por outras palavras, a curva só se define no segmento do meio. A influência dos pontos P_0 e P_3 assenta na determinação das direções tangentes de entrada e saída. Esta configuração torna-as inerentemente contínuas em 1º grau (isto é, garante-se que as derivadas são também contínuas), algo que não é possível dizer sobre as de Bézier - a sua travessia dos quatro pontos possibilita a definição de trajetos compostos por curvas tais que um mesmo ponto seja o fim duma curva e início de outra com tangentes discordantes.



A forma matricial da fórmula que avalia as coordenadas na curva para um t arbitrário é análoga a Bézier, mas com uma matriz característica M_{CR} diferente:

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \\ 1 & -\frac{5}{2} & 2 & -\frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} M_{\text{CR}} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{11}$$

$$P'(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \\ 1 & -\frac{5}{2} & 2 & -\frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} M_{\text{CR}} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{12}$$

1. Leitura e Conversão de Ficheiros .patch

Para cada patch do modelo, o ficheiro .patch disponibiliza uma lista de dezasseis índices dos vértices que lhe pertença. Sabendo isto, é possível construir três matrizes, P_{xyz}, para representar cada componente dos pontos de controlo envolvidos.

```
struct Patch {

  glm::mat4 MPMt_x;
  glm::mat4 MPMt_y;
  glm::mat4 MPMt_z;

  Patch(std::vector<glm::vec3>& controlPoints) {
    auto [Px, Py, Pz] = Pxyz(controlPoints);
    this->MPMt_x = bezier2::MPMt(Px);
    this->MPMt_y = bezier2::MPMt(Py);
    this->MPMt_z = bezier2::MPMt(Pz);
  }

  std::pair<glm::vec3, glm::vec3> evaluate(float u, float v) {
    const glm::vec4 U(u*u*u, u*u, u, 1);
    const glm::vec4 V(v*v*v, v*v, v, 1);
    const glm::vec4 dU(3*u*u, 2*u, 1, 0);
    const glm::vec4 dV(3*v*v, 2*v, 1, 0);
    glm::vec3 position = {
      glm::dot(U * MPMt_x, V),
      glm::dot(U * MPMt_y, V),
      glm::dot(U * MPMt_z, V),
    };
    glm::vec3 du = {
      glm::dot(dU * MPMt_x, V),
      glm::dot(dU * MPMt_y, V),
      glm::dot(dU * MPMt_z, V)
    };
    glm::vec3 dv = {
      glm::dot(U * MPMt_x, dV),
      glm::dot(U * MPMt_y, dV),
      glm::dot(U * MPMt_z, dV)
    };
    glm::vec3 normal = glm::normalize(glm::cross(du, dv));
    return { position, normal };
  }
};
```

```
namespace bezier2 {

  const glm::mat4 M(
    -1,  3, -3, 1,
     3, -6,  3, 0,
    -3,  3,  0, 0,
     1,  0,  0, 0
  );

  glm::mat4 MPMt(const glm::mat4& P) {
    return (M * P) * glm::transpose(M);
  }
};
```

A partir daqui, precalculam-se as matrizes de coeficientes constantes ao patch inteiro, MPM^T, usadas para uma avaliação eficiente das coordenadas (u, v) cujos valores incrementam consoante o nível de tesselação especificado em comando.

Assim, por cada quadrícula resultante da tesselação, seis vértices (dois por triângulo) com coordenadas nalguma combinação de $(u|(u+1), v|(v+1))$ são adicionados ao coletor em ordem antihorária.

1. Ecrã Inteiro

Tirando partido do que o GLUT aufere, acrescentou-se um modo de ecrã inteiro ao carregar no '0' do teclado. O funcionamento é intuitivo: o carregar da tecla despoleta a função abaixo, que guarda ou restaura as dimensões da janela consoante esteja ou não em preenchimento máximo do ecrã.

```
namespace window {
  bool inFullscreen = false;

  void toggleFullscreen() {
    static int wPrev = world.window.width;
    static int hPrev = world.window.height;
    static int xPrev = 100;
    static int yPrev = 100;

    if (inFullscreen == false) {
      wPrev = glutGet(GLUT_WINDOW_WIDTH);
      hPrev = glutGet(GLUT_WINDOW_HEIGHT);
      xPrev = glutGet(GLUT_WINDOW_X);
      yPrev = glutGet(GLUT_WINDOW_Y);
      inFullscreen = true;
      glutFullScreen();
    }
    else {
      glutPositionWindow(xPrev, yPrev);
      glutReshapeWindow(wPrev, hPrev);
      inFullscreen = false;
    }
  }
};
```

2. Tempo Entre Frames

A callback de renderização agora passa a chamar uma função que faz de "cronómetro", medindo quanto tempo se demora a trocar de frame. Esta medida de tempo é útil pois permite realizar transformações animadas, movimentar a câmara a uma velocidade independente da taxa de frames/segundo, e mesmo calcular esta última duma forma mais fiável.

```
namespace render {
  void renderScene(void) {

    clock::update();
    keybinds::update(clock::deltaTime);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    camera::set();

    if (axes::visible) axes::show();
    world.renderGroups(vboMode::withVBOs, clock::deltaTime);

    framesPerSecond::update(clock::currentTime, 100.0f);
    hud::show();

    glutSwapBuffers();
  }
};
```

```
namespace render{
  namespace clock {

    float currentTime = 0.0f;
    float lastTime = 0.0f;
    float deltaTime = 0.0f; // time between frames

    void update() {
      float lastTime = currentTime;
      currentTime = glutGet(GLUT_ELAPSED_TIME);
      deltaTime = (currentTime - lastTime) / 1000.0f;
      snprintf(infoString, sizeof(infoString), "Elapsed: % .1fs",
      currentTime / 1000.0f);
    }
  };
};
```

```
namespace render {
  namespace framesPerSecond {
    float lastSampleTime = 0.0f, frameCount = 0.0f, fps;

    void update(float currentTime, float samplingPeriod = 100.0f) {

      frameCount++;
      if (currentTime - lastSampleTime > samplingPeriod) {
        fps = frameCount * 1000.0f / (currentTime - lastSampleTime);
        lastSampleTime = currentTime;
        frameCount = 0;
        snprintf(infoString, sizeof(infoString), "Frames/s: %d", static_cast<int>(fps));
      }
    }
  };
};
```

A taxa de frames/segundo calcula-se à base de contagem de frames mostrados durante 100ms. Mal esse tempo seja ultrapassado, divide-se o número de frames contado pelo intervalo de tempo em segundos que passou.

Esta e mais outras estatísticas têm, cada uma, a sua string própria recorrentemente atualizada, que é mostrada por uma função de renderização de texto a ser abordada mais adiante.

3. Melhorias de Câmara e Controlos

A medida do tempo entre frames permitiu que se "libertasse" a câmara da sincronia com a renderização. Até à fase anterior, a câmara transladava/rodava 1 unidade fixa por cada frame após deteção de input (*frame-locked*). Isto sujeitava a velocidade do reposicionamento à potência do hardware.

Lembra um fenómeno visto em jogos mais antigos executados em máquinas modernas: certas animações notarem-se exageradamente rápidas ou lentas, por terem sido implementadas duma forma *frame-locked* sem expectativa de alguma vez serem testemunhadas nas taxas de frame mais comuns hoje em dia.

```cpp
namespace camera {
  namespace freeroam {
    ...
    void moveDown(float deltaTime) {
      pos -= worldUp * movementSpeed * deltaTime;
    }

    void rotateYaw(float direction, float deltaTime) {
      yaw += direction * sensitivity * deltaTime;
      updateVectors();
    }

    void rotatePitch(float direction, float deltaTime) {
      pitch += direction * sensitivity * deltaTime;
      if (pitch > 89.0f) pitch = 89.0f;
      if (pitch < -89.0f) pitch = -89.0f;
      updateVectors();
    }
  };
};
```

```cpp
namespace keybinds {
  std::unordered_set<unsigned char> keysPressed;
  std::unordered_set<int>           specialKeysPressed;
  ...
  void keyboardSpecialUp(int key_code, int x, int y) {
    specialKeysPressed.erase(key_code);
  }

  void keyboardUp(unsigned char key, int x, int y) {
    keysPressed.erase(key);
  }

  void keyboardSpecial(int key_code, int x, int y) {
    specialKeysPressed.insert(key_code);
  }

  void keyboard(unsigned char key, int x, int y) {
    if (!toggleKeys.contains(key))
      keysPressed.insert(key);

    switch (key) {...}
  }

  void update(float deltaTime) {
    // Handle continuous key presses
    for (unsigned char key : keysPressed) {
      camera::handleKey(key, deltaTime);
    }

    for (int key_code : specialKeysPressed) {
      camera::handleKeySpecial(key_code, deltaTime);
    }
  }
};
```

Os conjuntos *keysPressed* e *specialKeysPressed* ajudam a manter um estado global das teclas conseguiu-se não só passar a controlar a câmara a uma cadência assíncrona da renderização, como também usar várias teclas ao mesmo tempo.

4. Renderização de Texto

I. Modificar referencial da projeção (*GL_PROJECTION*):

- Trocar para um região de visualização ortonormada em duas dimensões - pois para este HUD textual não é relevante uma profundidade em Z.
- Para *gluOrtho2D* são passados valores que definam os limites da região nos lados, em cima e em baixo. Neste caso, como só há um único viewport, servem os limites da janela completa.

II. Modificar referencial dos modelos (*GL_MODELVIEW*):

- Após escolher a cor do texto, usar *glRasterPos* para declarar a posição rasterizada inicial do texto passado por argumento - isto é, em que coordenadas relativas do canto superior esquerdo da janela se desenhará o primeiro caracter.

Feito tudo isto, restauram-se os referenciais e estados de atributo prévios às edições.

```
namespace render {
  namespace hud {
    void show() {

      glPushAttrib(GL_CURRENT_BIT | GL_ENABLE_BIT);

      double windowWidth = glutGet(GLUT_WINDOW_WIDTH);
      double windowHeight = glutGet(GLUT_WINDOW_HEIGHT);

      glMatrixMode(GL_PROJECTION); glPushMatrix();
      glLoadIdentity();
      gluOrtho2D(0, windowWidth, 0, windowHeight);

      glMatrixMode(GL_MODELVIEW); glPushMatrix();
      glLoadIdentity();

      std::vector<std::string> lines = {...};

      glColor3f(1.0f, 1.0f, 1.0f);

      float y = windowHeight - 25.0f;
      for (const auto& line : lines) {
        glRasterPos2f(10, y);
        for (char c : line) {
          glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, c);
        }
        y -= 15.0f;
      }

      glMatrixMode(GL_PROJECTION); glPopMatrix();
      glMatrixMode(GL_MODELVIEW); glPopMatrix();

      glPopAttrib();
    }
  };
};
```

5. Vertex Buffer Objects

Até esta fase, cada referência a um modelo no ficheiro config.xml despoletava um carregamento dos seus vértices para a memória. Isto é ineficiente especialmente no caso do sistema solar, visto que é praticamente todo composto pelo mesmo modelo de esfera.

Em âmbito experimental, antes de tentar o carregamento para VBOs, decidiu-se implementar um meio-termo: a definição dum coletor unitário que guardasse os vértices dos modelos uma só vez e os desenhasse por chamada dum grupo hierárquico com o nome do ficheiro como referência.

Mais tarde comparou-se o desempenho da renderização por VBOs versus essa estratégia intermediária. Com a cena do sistema solar, assistiu-se a um ganho significativo de FPS (cerca de 100-150), mas introduzir a renderização de texto - que ainda é em modo imediato - reduziu bastante esse ganho (até 50 no melhor caso).

```
struct ModelStorage {

  std::unordered_map<std::string, Model> byFilename = {};

  const Model& get(const std::string& filename) {
    return byFilename.at(filename);
  }

  bool contains(const std::string& filename) const {
    return byFilename.find(filename) != byFilename.end();
  }

  void tryLoad(std::string& filename) {...}

  void drawAll(bool withVBO, std::vector<std::string> filenames) {...}

  void initializeAllVBOs() {...}

  void cleanupAllVBOs() {...}

  ModelStorage() = default;

  ModelStorage(std::vector<std::string> modelFilenames) {
    for (auto& filename : modelFilenames) {
      tryLoad(filename);
    }
  }
};
```

Passos para o desenho dum modelo via VBO:

0. Garantir que foi inicializada a biblioteca GLEW antes de qualquer interação, para evitar erros de falta de permissões.

I. Inicialização do Buffer:

1. Usar *glGenBuffers* para obter um identificador livre para o VBO que alojará os vértices, guardando esse identificador numa variável apropriada (aqui *vboID*).
2. Usar *glBindBuffer* para estabelecer ligação entre o buffer do identificador passado e o alvo *GL_ARRAY_BUFFER*, para a placa gráfica o interpretar especificamente como um buffer de vértices. Um alvo não consegue estar ligado a múltiplos buffers em simultâneo - logo, qualquer ligação *a priori* que tenha será quebrada para estabelecer esta.
3. Usar *glBufferData* para enviar os dados dos vértices para o VBO atualmente ligado ao alvo *GL_ARRAY_BUFFER*. Neste caso, os vértices estão definidos num vetor de floats, em que cada terno de floats corresponde às componentes x, y, z de cada um. Assim, passa-se por argumento que o número de vértices é um terço do tamanho do vetor. O argumento *GL_STATIC_DRAW* indica que os dados estão a ser carregados esta única vez e serão para uso recorrente daqui em diante.

II. Desenho do Conteúdo do Buffer:

1. Usar *glBindBuffer* para garantir que o VBO do modelo pretendido é o que está atualmente ligado.
2. Usar *glEnableClientState* para indicar que se pretende aceder aos dados do VBO atualmente ligado. A terminologia vem do facto de que, neste contexto, este programa (cliente, a rodar no processador) solicita acesso a dados da placa gr+afica (servidor).
3. Usar *glVertexPointer* para desenhar os vértices, especificando que têm cada um 3 coordenadas, do tipo float, e como tendo 0 elementos com dados extra (outrora poderiam haver elementos com informação de cor, normal, ou mapeamento de textura). O último argumento é um apontador relativo para o início do buffer.
4. Usar *glDisableClientState* para indicar ao GPU que acabou a necessidade de acesso aos dados do VBO.
5. Usar *glBindBuffer* com identificador 0 para desligar o VBO que se acabou de utilizar. Evitar a persistência da ligação colmata a possiblidade de acessos erróneos ao VBO fora do seu contexto de uso pretendido.

III. Libertação dos Dados do Buffer:

1. Usar glDeleteBuffers com o identificador do VBO para explicitamente limpar os seus dados e voltar a dar o identificador como livre para uso futuro. É uma boa prática de se tomar no fim do programa.

```
struct Model {

  std::string filename;
  std::vector<float> vertices;
  bool showAxes = false;
  unsigned int vertexCount = 0;
  unsigned int vboID = 0;

  Model(const std::string filename) : filename(filename) {
    vertices = modelFileManagement::loadModelVertices(filename);
  }

  void drawVBO() const {
    if (showAxes) drawLocalAxes();

    glBindBuffer(GL_ARRAY_BUFFER, vboID);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, 0);

    glDrawArrays(GL_TRIANGLES, 0, vertexCount);

    glDisableClientState(GL_VERTEX_ARRAY);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

  }

  void drawImmediate() const { ... }

  void initializeVBO() {
    glGenBuffers(1, &vboID);
    glBindBuffer(GL_ARRAY_BUFFER, vboID);
    glBufferData(
      GL_ARRAY_BUFFER,
      vertices.size() * sizeof(float),
      vertices.data(),
      GL_STATIC_DRAW
    );
    vertexCount = vertices.size() / 3;
  }

  void deleteVBO() {
    if (vboID != 0) {
      glDeleteBuffers(1, &vboID);
      vboID = 0;
    }
  }
};
```

```
int main(int argc, char** argv) {
  world = configParser::loadWorld("config.xml");
  std::cout << std::string(world) << std::endl;
  glutInit(&argc, argv);
  ...
  // Initialize GLEW
  GLenum err = glewInit();
  if (GLEW_OK != err) {
    fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
    return 1;
  }

  world.modelStorage.initializeAllVBOs();
  atexit([]() { world.modelStorage.cleanupAllVBOs(); });
  ...
  glutMainLoop();

  return 1;
}
```

6. Rotações Animadas

Foi acrescentada uma transformação de rotação cujo ângulo incrementado se deduz pelo Δt entre frames passado como argumento. Assim, quanto mais tempo o frame seguinte demora a ser desenhado, mais ângulo é aplicado à matriz do grupo. Apesar de ter sido mantida a rotação estática de fases anteriores, é tecnicamente possível usar esta para o mesmo efeito - basta instanciar uma rotação animada tal que $t_{\text{Period}} = 0$.

Verifique-se que os incrementos a t são normalizações de Δt, para garantir proporcionalidade perante t_{Period}.

```
struct AnimatedRotation {

  glm::vec3 axis = {0.0f,1.0f,0.0f};
  float t = 0.0f;
  float tPeriod = 0.0f;

  AnimatedRotation(glm::vec3 axis, float tPeriod) : axis(axis), tPeriod(tPeriod) {}

  void apply(float tDelta) {
    glRotatef(360.0f * t, axis.x, axis.y, axis.z);
    auto tDeltaNormalised = tDelta / tPeriod;
    t = (t + tDeltaNormalised <= 1.0f) ? t + tDeltaNormalised : 0.0f;
  }
};
```

7. Translações Animadas

A translação animada baseia-se no cálculo de $p(t)$ e $p'(t)$ para a spline Catmull-Rom composta definida pelos pontos de controlo passados. O repetir dos três pontos iniciais na cauda do vetor vem do facto de que, numa spline desta categoria com N pontos de controlo, não há interpolação para os segmentos (P_0, P_1) e $(P_{N\text{-}1}, P_N)$. Assim, há que compensar com a declaração de dois segmentos contíguos tanto entre si como entre o último e o primeiro pré-existentes (para formar um circuito fechado). É aqui que nota a importância das curvas Catmull-Rom serem inerentemente contínuas em 1º grau.

```
namespace catRom {
  struct Spline {
    std::vector<glm::mat3x4> segmentMPs = {};

    std::pair<glm::vec3, glm::vec3> evaluate(float t) const {
      if (segmentMPs.empty()) return { glm::vec3(0), glm::vec3(0) };
      float tScaled  = t * segmentMPs.size();
      float tSegment = glm::fract(tScaled);
      int   iSegment = glm::min((int)tScaled, (int)segmentMPs.size() -
      1);
      return point(segmentMPs[iSegment], tSegment);
    }

    void transform(float t, float aligned, glm::vec3 worldUp) {
      const auto [p, dp] = evaluate(t);

      glMultMatrixf(glm::value_ptr(glm::mat4(
        glm::vec4(1, 0, 0, 0),
        glm::vec4(0, 1, 0, 0),
        glm::vec4(0, 0, 1, 0),
        glm::vec4(p,       1)
        )));

      if (aligned) {
        auto front = glm::normalize(dp);
        auto right = glm::normalize(glm::cross(front,worldUp));
        auto up = glm::normalize(glm::cross(right,front));

        glMultMatrixf(glm::value_ptr(glm::mat4(
          glm::vec4(front,   0),
          glm::vec4(up,      0),
          glm::vec4(right,   0),
          glm::vec4(0, 0, 0, 1)
          )));
      }
    }
};
```

```
struct AnimatedTranslation {

  static inline const int tessellationLevels[4] = { 1, 2, 4, 10 };
  static inline int currentTessIndex = -1;
  static inline bool showPath = false;

  static inline glm::vec3 worldUp;
  std::vector<glm::vec3> controlPoints = {};
  catRom::Spline crPath;

  float t = 0.0f;
  float tPeriod = 0.0f;
  bool aligned = true;

  AnimatedTranslation(std::vector<glm::vec3> controlPoints, float
  tPeriod, bool isAligned) :
    tPeriod(tPeriod),
    aligned(isAligned)
  {
    this->controlPoints = controlPoints;
    // to close the loop
    (this->controlPoints).push_back(controlPoints[0]);
    (this->controlPoints).push_back(controlPoints[1]);
    (this->controlPoints).push_back(controlPoints[2]);

    crPath = catRom::Spline(this->controlPoints);
  }

  void drawControlPoints() {...}

  static void nextTessellationLevel() {
    currentTessIndex = (currentTessIndex + 1) %
    (sizeof(tessellationLevels) / sizeof(int));
  }

  void apply(float tDelta) {

    if (showPath) drawControlPoints();

    crPath.drawWhole(tessellationLevels[currentTessIndex]);
    crPath.transform(t, aligned, worldUp);

    auto tDeltaNormalised = tDelta / tPeriod;
    t = (t + tDeltaNormalised <= 1.0f) ? t + tDeltaNormalised : 0.0f;
  }
```

Para alinhar um modelo ao longo da tangente (direção frontal, assumida no eixo X) da sua posição t numa spline composta por n segmentos, foi usado o raciocínio do algoritmo Gram-Schmidt já presente na ortonormalização do referencial da câmara em 1ª pessoa.

Considere-se t_{g} o progresso ao longo da spline inteira e t_s o progresso ao longo do atual segmento:

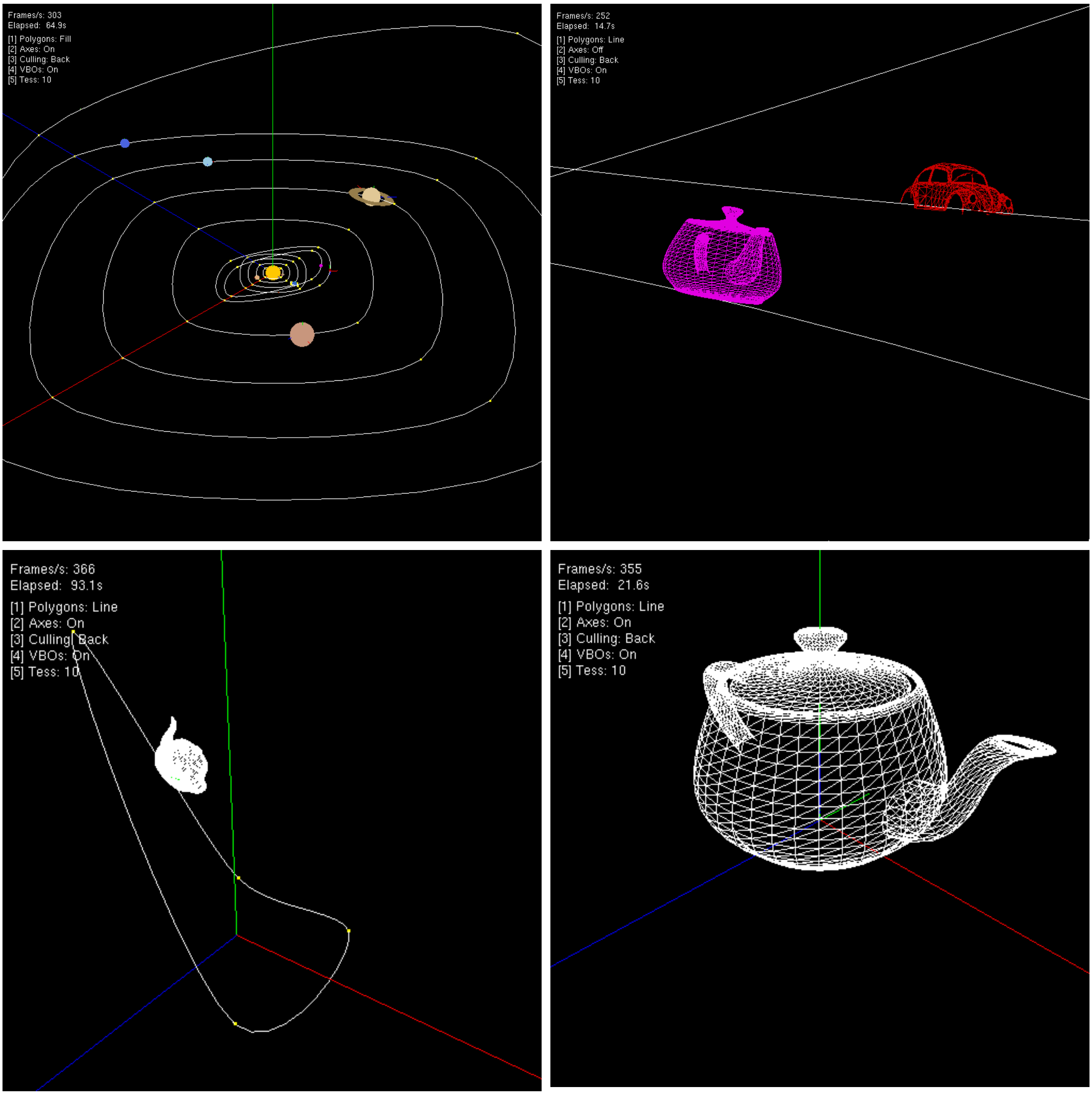
$$t_s = \{n \cdot t_{\mathrm{g}}\} = n \cdot t_{\mathrm{g}} - \lfloor n \cdot t_{\mathrm{g}} \rfloor \tag{13}$$

$$\begin{aligned} \mathrm{X} &= P'(t_s) \\ \mathrm{Z} &= X \times Y_{\mathrm{global}} \\ \mathrm{Y} &= Z \times X \end{aligned} \tag{14}$$

$$\mathrm{Rot} = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

8. Demonstração

Para a cena do sistema solar, colocou-se os planetas, a Lua, e dois asteróides em translação animada. Os asteróides são o omnipresente bule de Newell e o carocha de Sutherland - mais um modelo pioneiro na computação gráfica. Este último foi alvo duma translação estática adicional no ficheiro XML, visto que tem um certo desvio à origem no eixo Y.



Sistema solar, meteoros, ficheiros "test_3_1.xml" e "test_3_2.xml"