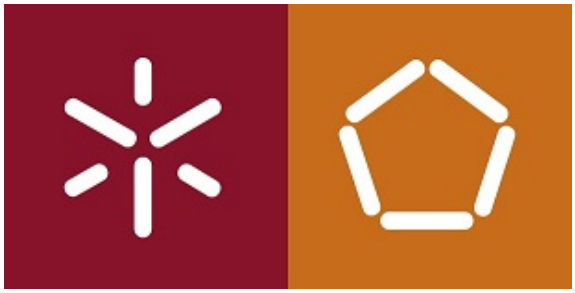
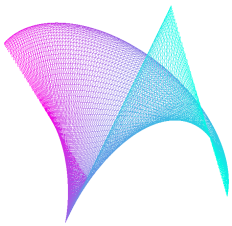


Universidade do Minho
Departamento de Informática

Solaris



Computação Gráfica
Grupo 24 - Fase 3

1 de maio, 2022


Beatriz Rodrigues
(*a93230*)


Francisco Neves
(*a93202*)


Guilherme Fernandes
(*a93216*)


João Carvalho
(*a93166*)

Índice

1. Introdução

O programa desenvolvido, *Solaris*, foi atualizado de forma a implementar melhorias. Uma delas consistiu em garantir que os desenhos são efetuados através da utilização de *VBO's*. Para além disso, de forma a conferir movimento e animação aos corpos celestes, foram implementadas curvas de *Catmull-Rom*. Assim, os corpos celestes, passam então a ter um movimento de translação e de rotação. A *script* utilizada para gerar a *scene* correspondente ao Sistema Solar foi, também ela, atualizada devido às alterações requeridas, tendo em conta as novas melhorias do programa desenvolvido.

Por outro lado, foram ainda utilizados *patches* de Bezier para a criação de uma nova primitiva no *generator*.

Por fim, foram feitas algumas melhorias no movimento da câmara, adicionadas mais *keybinds* e iniciou-se a utilização da biblioteca *ImGUI* que disponibiliza a visualização de janelas com informação interativa.

2. Engine

2.1 Conversão para *VBOs*

Nas fases anteriores, os modelos pretendidos eram desenhados de modo imediato, no entanto, nesta fase foi feita a transição para a utilização de *VBOs* (*Vertex Buffer Objects*). Estes podem ser vistos como um *array* residente na memória da placa gráfica e permitem melhorias significativas no que toca à performance do programa.

Foi criada uma nova classe denominada `ModelBuffer` de forma a permitir a gestão dos *VBOs*. Esta contém o `id` do *buffer* (`_vbo`) e o número de vértices do mesmo (`_n_vertices`).

Esta classe dispõe de um método para carregar os vértices de um ficheiro `.3d` para um vetor e, posteriormente, para um *buffer*.

```
auto ModelBuffer::try_from_file(std::string_view file_path) noexcept
    -> cpp::result<ModelBuffer, ParseError>
{
    std::ifstream file_stream(file_path.data());
    if (!file_stream.is_open()) {
        return cpp::fail(ParseError::PRIMITIVE_FILE_NOT_FOUND);
    }
    float x, y, z;
    auto vertices = std::vector<float>();
    while (file_stream >> x >> y >> z) {
        vertices.push_back(x);
        vertices.push_back(y);
        vertices.push_back(z);
    }
    file_stream.close();

    GLuint vbo;
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(
        GL_ARRAY_BUFFER,
        sizeof(float) * vertices.size(),
        vertices.data(),
        GL_STATIC_DRAW
    );

    return ModelBuffer(vbo, vertices.size());
}
```

Existe ainda um método para dar *bind* e desenhar o *buffer* em questão.

```
auto ModelBuffer::draw() const noexcept -> void {
    glBindBuffer(GL_ARRAY_BUFFER, _vbo);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, _n_vertices);
}
```

Uma vez que na fase anterior do projeto foi otimizada a leitura de modelos para cada ficheiro `.3d` ser lido uma única vez e o seu vetor de pontos partilhado por todos os modelos que o utilizam, nesta fase esse processo foi alterado de forma a partilhar um `ModelBuffer` em vez de um `std::vector<Point>`.

2.2 Transformações com tempo

Para animar as cenas criadas, foram implementadas transformações que ocorrem ciclicamente num intervalo de tempo.

Para manter a estrutura de transformações da fase anterior, o método `apply` das mesmas passa a receber o tempo que a simulação esteve a correr (este tempo não é igual ao tempo que passou desde o início do programa, pois a simulação pode ser acelerada, retardada, ou até mesmo parada).

2.2.1 Rotação

De forma a suportar rotações com tempo foi criada uma nova classe (`TimedRotation`) que contém um parâmetro que indica o tempo que um determinado objeto deve demorar a efetuar uma rotação completa (`_time`).

Recorrendo ao tempo de simulação decorrido, é possível efetuar o cálculo do ângulo no instante pretendido.

```
auto TimedRotation::apply(float elapsed_time) const noexcept -> void {
    glRotatef(
        elapsed_time * (_time ? 360.0f / _time : 0),
        _coords.x(), _coords.y(), _coords.z()
    );
}
```

2.2.2 Translação

Uma vez que as translações utilizam curvas de *Catmull-Rom* e a única variante ao trabalhar diiferentes tipos de curvas é a matriz que as descreve, foi desenvolvida uma classe (`Curve`) genérica, de forma a ser possível trabalhar com qualquer tipo de curva, fornecendo assim um maior grau de abstração ao programa.

Esta classe contém uma referência para a matriz de geometria que usa (como esta nunca é alterada, assim pode ser partilhada por todas as instâncias), um vetor com os pontos que descrevem a curva e dois construtores, um para as curvas de *Catmull-Rom* e outro para as curvas de *Bezier*

```
static const std::array<std::array<float, 4>, 4> catmull_rom_matrix{{
    {-0.5f, +1.5f, -1.5f, +0.5f},
    {+1.0f, -2.5f, +2.0f, -0.5f},
    {-0.5f, +0.0f, +0.5f, +0.0f},
    {+0.0f, +1.0f, +0.0f, +0.0f},
}};

static const std::array<std::array<float, 4>, 4> bezier_matrix{{
    {-1.0f, +3.0f, -3.0f, +1.0f},
    {+3.0f, -6.0f, +3.0f, +0.0f},
    {-3.0f, +3.0f, +0.0f, +0.0f},
    {+1.0f, +0.0f, +0.0f, +0.0f},
}};

auto Curve::catmull_rom(std::vector<Point> points) -> Curve {
    return Curve(catmull_rom_matrix, std::move(points));
```

```
}

auto Curve::bezier(std::vector<Point> points) -> Curve {
    return Curve(bezier_matrix, std::move(points));
}
```

Além disso, esta classe dispõe de um método que, recebendo um instante de tempo, calcula a posição na curva relativa ao mesmo. Inicialmente é calculado o segmento da curva em que se encontra o instante de tempo, assim como o instante de tempo nesse segmento.

```
auto t = global_time * _points.size();
auto segment = (int) floor(t);
t -= segment;
```

Após isso é calculado o índice do primeiro vértice e os pontos pertencentes à curva do instante de tempo.

```
auto fst_idx = segment + _points.size() - 1;

auto p1 = _points[(fst_idx + 0) % _points.size()];
auto p2 = _points[(fst_idx + 1) % _points.size()];
auto p3 = _points[(fst_idx + 2) % _points.size()];
auto p4 = _points[(fst_idx + 3) % _points.size()];

const std::array<std::array<float, 4>, 3> p{{
    {p1.x(), p2.x(), p3.x(), p4.x()},
    {p1.y(), p2.y(), p3.y(), p4.y()},
    {p1.z(), p2.z(), p3.z(), p4.z()},
}};
```

Com a matriz dos pontos podemos então calcular a posição e o vetor tangente do ponto no instante de tempo indicado.

```
std::array<float, 3> pv{};
std::array<float, 3> dv{};

for (size_t i = 0; i < 3; ++i) {
    std::array<float, 4> a{};

    for (size_t j = 0; j < 4; ++j) {
        for (size_t k = 0; k < 4; ++k) {
            a[j] += p[i][k] * _geometry_matrix[j][k];
        }
    }

    for (size_t j = 0; j < 4; j++) {
        pv[i] += tv[j] * a[j];
        dv[i] += tvd[j] * a[j];
    }
}

return {
    Point::cartesian(pv[0], pv[1], pv[2]),
    Point::cartesian(dv[0], dv[1], dv[2]),
};
```

Foi então criado um novo tipo de transformação, que trata deste tipo de translações (`TimedTranslation`). Esta, quando criada, recebe o vetor de pontos que descrevem a curva de *Catmull-Rom*, o tempo de rotação e uma *flag* a indicar se o objeto está alinhado, isto é se o objeto roda no sentido do seu movimento, e cria um vetor com um número pré-definido de Pontos, que representam a curva que descreve o movimento, utilizando o método `Curve::get_position` descrito anteriormente. Assim as trajetórias dos objetos podem ser desenhadas sem ser necessário efetuar o seu cálculo a todos os *frames*, o que provocaria uma grande ineficiência ao programa.

```
TimedTranslation::TimedTranslation(
    std::vector<Point> points,
    float time,
    bool is_aligned
)
    : _curve(Curve::catmull_rom(std::move(points)))
    , _trajectory(std::vector<Point>())
    , _time(time)
    , _is_aligned(is_aligned)
    , _prev_y(Point::cartesian(0, 1, 0))
{

    for (auto t = 0.0f; t < 1.0f; t += 1.0f / TESS) {
        _trajectory.push_back(_curve.get_position(t).first);
    }
}
```

Tal como as outras transformações, esta classe, contém um método `apply`. Caso o objeto esteja alinhado com a curva, são calculados os eixos da rotação a partir do vetor tangente à curva na posição atual.

```
auto TimedTranslation::apply(float elapsed_time) const noexcept -> void {
    auto [pos, dir] =  _curve.get_position(elapsed_time / _time);

    glTranslatef(pos.x(), pos.y(), pos.z());

    if (_is_aligned) {
        auto x = dir.normalize();
        auto z = Point(x).cross(_prev_y).normalize();
        auto y = Point(z).cross(x).normalize();
        glMultMatrixf(build_rotation_matrix(x, y, z).data());
    }
}

auto TimedTranslation::build_rotation_matrix(
    Point x, Point y, Point z
) const noexcept -> std::array<float, 16> {
    return std::array<float, 16>{{
        x.x(), x.y(), x.z(), 0,
        y.x(), y.y(), y.z(), 0,
        z.x(), z.y(), z.z(), 0,
            0,     0,     0, 1,
    }};
}
```

2.3 Movimento

Para melhorar o movimento da câmara foi utilizado um *array* onde são guardadas as teclas pressionadas. Este *array* é passado por referência em cada *frame* para as funções da câmara que a movimentam, oferecendo, desta forma, um movimento mais controlado.

```
void handle_key_down(unsigned char key, int x, int y) {
    state::keyboard[key] = true;
}

void handle_key_up(unsigned char key, int x, int y) {
    state::keyboard[key] = false;
}

auto Camera::react_key(
    std::array<bool, std::numeric_limits<unsigned char>::max()>& kb
) noexcept -> void {
    switch (_mode) {
    case CameraMode::ORBIT:
        react_key_orbit(kb);
        break;
    case CameraMode::FPV:
        react_key_fpv(kb);
        break;
    default:
        break;
    }
}

auto Camera::react_key_orbit(
    std::array<bool, std::numeric_limits<unsigned char>::max()>& kb
) noexcept -> void {
    auto beta = _eye.beta();
    auto radius = _eye.radius();
    auto alpha = _eye.alpha();

    if (kb['w']) beta += 0.1f;
    if (kb['s']) beta -= 0.1f;
    if (kb['a']) alpha -= 0.1f;
    if (kb['d']) alpha += 0.1f;
    if (kb['+']) radius -= 0.5f;
    if (kb['-']) radius += 0.5f;

    beta = std::clamp(beta, -1.5f, 1.5f);

    _eye = Point::spherical(radius, alpha, beta);
}

auto Camera::react_key_fpv(
    std::array<bool, std::numeric_limits<unsigned char>::max()>& kb
```

```
) noexcept -> void {
    auto vec = _center - _eye;
    vec.normalize();

    auto mov = Point::cartesian(0.f, 0.f, 0.f);

    if (kb['w']) mov += vec * 0.5f;
    if (kb['s']) mov -= vec * 0.5f;
    if (kb['a']) mov -= Point(vec).cross(_up).normalize() * 0.5f;
    if (kb['d']) mov += Point(vec).cross(_up).normalize() * 0.5f;
    if (kb['+']) mov += Point(_up) * 0.5f;
    if (kb['-']) mov -= Point(_up) * 0.5f;

    _eye += mov;
    _center += mov;
}
```

2.4 *Keybinds*

Foram adicionadas algumas *keybinds* que alteram o modo de visualização ou afetam a simulação. São elas:

- `F1`: alternar entre modos de câmara;
- `F2`: desenhar linhas e curvas;
- `F3`: alternar entre modos de desenho de polígonos (`GL_FILL` ou `GL_LINE`);
- `F10`: retardar a simulação;
- `F11`: pausar a simulação;
- `F12`: acelerar a simulação.

2.5 `ImGUI`

Com o objetivo de adicionar funcionalidades ao *engine* foi inicializada a utilização da biblioteca `ImGUI`.

Embora nesta fase o projeto apenas possua um pequeno menu com opções, é algo que pretendemos expandir no futuro, permitindo assim a oferta opções de novas opções que sejam consideradas relevantes e úteis no contexto do projeto, tal como o carregamento de ficheiros `XML`.

3. *Generator*

Nesta fase foi implementada a capacidade do *generator* interpretar *patches* de *Bezier* e convertê-los num ficheiro `.3d` contendo triângulos.

3.1 Leitura do ficheiro `.patch`

Para gerar o patch, primeiramente tem de ser efetuada a leitura do ficheiro `.patch` sendo o seu conteúdo guardado. Para isso, começa-se por efetuar a leitura do número de *patches* presentes e, em seguida, são lidos e armazenaods esses *patches* num vetor (cada *patch* é representado por um *array* de 16 elementos).

```
size_t num_patches;
file_stream >> num_patches;
auto patches = std::vector<std::array<size_t, 16>>(num_patches);
for (size_t i = 0; i < num_patches; ++i) {
    for (size_t j = 0; j < 16; ++j) {
        size_t idx;
        file_stream >> idx;
        file_stream.ignore();
        patches[i][j] = idx;
    }
}
```

Um elemento de um *patch* é um índice que representa um ponto. Posteriormente, é efetuada a leitura e o armazenamento desses pontos num vetor.

```
size_t num_points;
file_stream >> num_points;
auto points = std::vector<Point>(num_points);
for (size_t i = 0; i < num_points; ++i) {
    float x, y, z;
    file_stream >> x;
    file_stream.ignore();
    file_stream >> y;
    file_stream.ignore();
    file_stream >> z;
    file_stream.ignore();

    points[i] = Point::cartesian(x, y, z);
}
```

Com os *patches* e os seus pontos armazenados, segue-se para o cálculo das coordenadas do objeto `.3d`, partindo de um nível de tesselação à escolha do utilizador.

Inicialmente, considerou-se a reutilização da função `Curve::get_position` utilizada no `engine` de forma a obter o ponto na curva para cada `u`, criando assim uma nova curva com a qual seria obtido o ponto na posição `v`. No entanto, isto implicaria fazer uma grande quantidade de multiplicações de matrizes para todos os pontos do objeto, logo foi decidida a utilização da seguinte fórmula simplificada:

$$p(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Onde a seguinte matriz pode ser pré-calculada:

$$M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T$$

Assim, foi desenvolvido o seguinte código:

```
auto get_patch_point(
    std::array<std::array<Point, 4>, 4>& patch, float u, float v
) noexcept -> Point {
    std::array<std::array<Point, 4>, 1> temp{};
    matrices::mult<float, Point, Point, 1, 4, 4>(
        {{{u*u*u, u*u, u, 1}}},
        patch,
        temp
    );

    std::array<std::array<Point, 1>, 1> point{};
    matrices::mult<Point, float, Point, 1, 4, 1>(
        temp,
        {{{v*v*v}, {v*v}, {v}, {1}}},
        point
    );
    return point[0][0];
}
```

```
auto coords = std::vector<Point>();
for (size_t i = 0; i < num_patches; ++i) {
    auto& curr_patch = patches[i];
    auto patch_mat = Curve::patch_matrix({{
        {
            points[curr_patch[0]],
            points[curr_patch[1]],
            points[curr_patch[2]],
            points[curr_patch[3]]
        },
        {
            points[curr_patch[4]],
            points[curr_patch[5]],
            points[curr_patch[6]],
            points[curr_patch[7]]
        },
        {
            points[curr_patch[8]],
            points[curr_patch[9]],
            points[curr_patch[10]],
            points[curr_patch[11]]
        },
        {
            points[curr_patch[12]],
            points[curr_patch[13]],
            points[curr_patch[14]],
            points[curr_patch[15]]
        },
    }});
    for (size_t ui = 0; ui < tess; ++ui) {
        auto u = ui / static_cast<float>(tess);
        auto nu = (ui + 1) / static_cast<float>(tess);

        for (size_t vi = 0; vi < tess; ++vi) {
            auto v = vi / static_cast<float>(tess);
            auto nv = (vi + 1) / static_cast<float>(tess);

            Point p0 = get_patch_point(patch_mat, u, v);
            Point p1 = get_patch_point(patch_mat, u, nv);
            Point p2 = get_patch_point(patch_mat, nu, v);
            Point p3 = get_patch_point(patch_mat, nu, nv);

            coords.push_back(p3);
            coords.push_back(p2);
            coords.push_back(p1);

            coords.push_back(p2);
            coords.push_back(p0);
            coords.push_back(p1);
        }
```

```
    }
}
```

O método `Curve::patch_matrix` realiza as multiplicações de matrizes mais exigentes, efetuando-o uma única vez para cada *patch*.

4. Sistema Solar

De forma a complementar a *demo scene* da fase anterior, foram ainda adicionadas diversas novas *features* ao Sistema Solar produzido, nomeadamente:

1. Órbitas de Planetas, Asteróides e do Cometa *Halley*;
2. Rotações de Planetas;
3. Adição do Cometa *Halley* representado por um *teapot*;
4. Inclinação de Planetas;
5. Adição do anel de Úrano.

Para permitir a automização na criação do ficheiro `XML`, todas estas *features* são escritas no ficheiro pretendido recorrendo à *script* em `Python` criada na fase anterior. Esta foi também passível de melhorias de forma a implementar aquilo que lhe é pretendido.

Por fim, de forma a automatizar o cálculo das órbitas dos planetas recorrendo aos pontos de uma curva de *Catmull-Rom*, foi desenvolvida uma classe em `Python` denominada `CurvePoints` que recebe como parâmetros o número de divisões que a curva deve conter e o raio do círculo que a curva deve formar. Posteriormente, recorrendo ao método `calc_points()` desta classe é obtida uma lista de dicionários indicando as coordenadas de cada ponto pretendido:

```
import math

class CurvePoints:
    def __init__(self, n_divs, radius):
        self.n_divs = n_divs
        self.radius = radius

    def calc_points(self):
        step = int(90 / self.n_divs)
        list_points = []
        for alpha in range(360, 0, -step):
            x = self.radius * math.cos(math.radians(alpha))
            z = self.radius * math.sin(math.radians(alpha))
            list_points.append({"x":x, "y":0, "z":z})
        return list_points
```

5. Resultados Obtidos

As imagens seguintes ilustram os resultados obtidos.

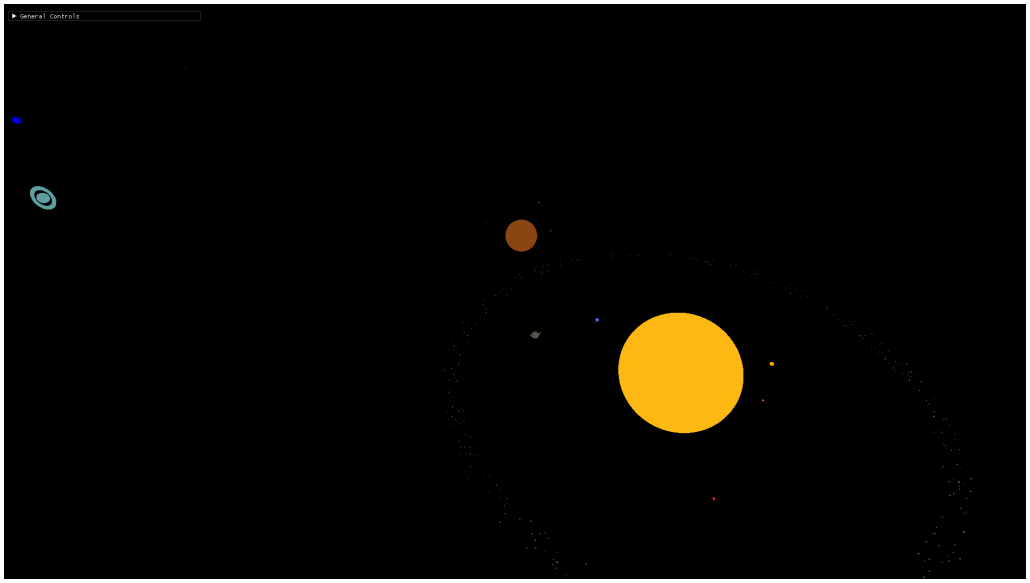


Figura 5.1: Sistema Solar

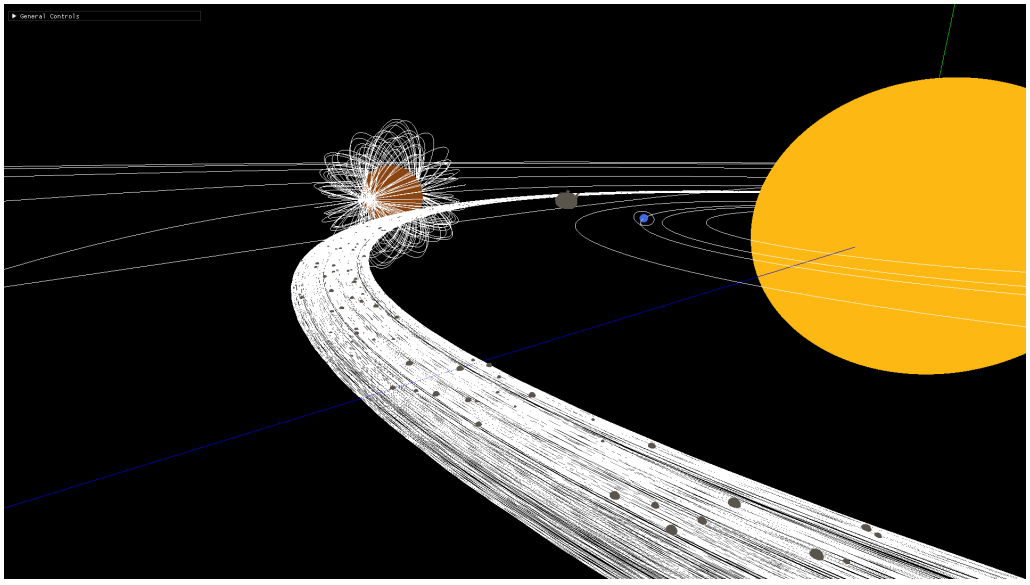


Figura 5.2: Curvas de *Catmull-Rom* na Cintura de Asteróides

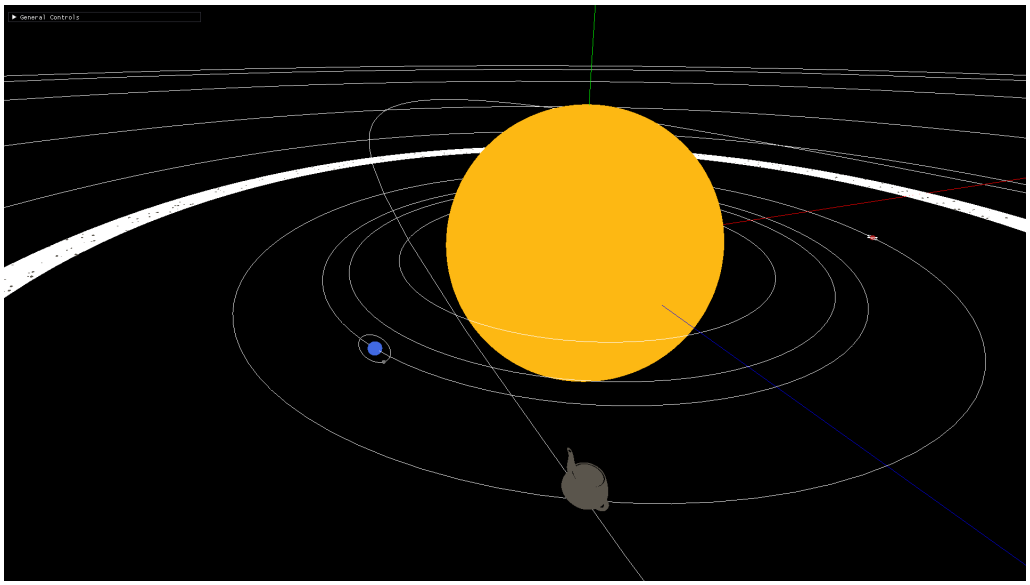


Figura 5.3: Curvas de *Catmull-Rom* nos diversos astros

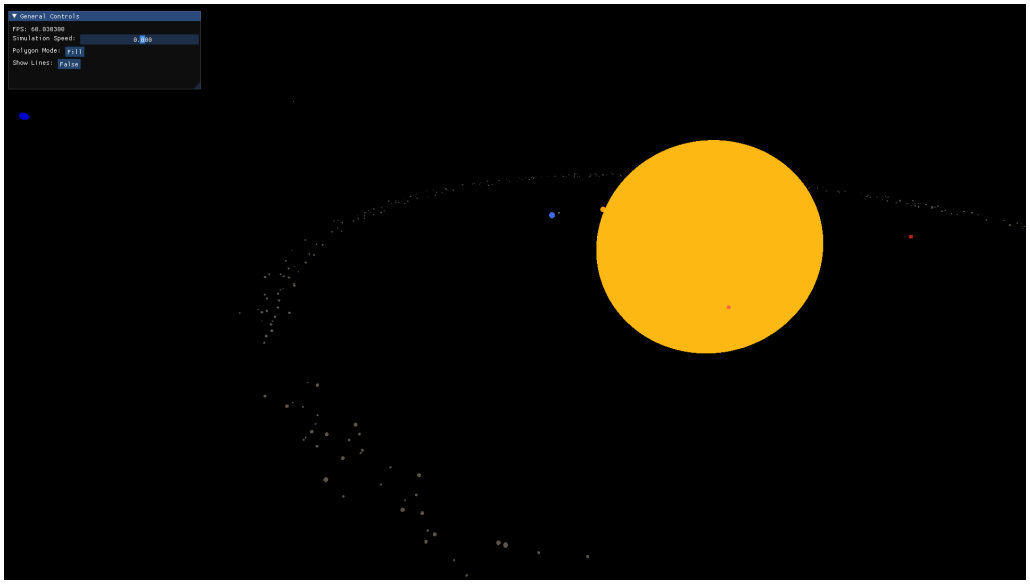


Figura 5.4: Integração da biblioteca `ImGUI`

6. Conclusão

Em suma, considera-se que foram adicionadas e melhoradas bastantes funcionalidades ao *Solaris* e que estas foram implementadas através de código expansível e eficiente.

Apesar disso, no futuro pretende-se ainda expandir a incorporação da biblioteca `ImGUI`, assim como a implementação de terrenos complexos gerados a partir de uma imagem.

Por fim, seria também interessante a adição de novas *scenes* para além da do Sistema Solar.