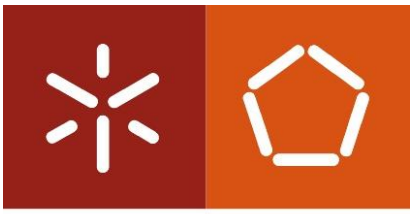


Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica - 3ª Fase

Grupo 14

Luís Faria (A93209) Joaquim Roque (A93310)
Gabriela Prata (A93288)

Ano Lectivo 2021/2022



0.1 Introdução

Na terceira fase pretende-se que, uma vez mais, se amplie o funcionamento das aplicações desenvolvidas anteriormente. Contrariamente ao que havia sido feito até aqui, é requirido que se usem *vertex buffer objects (VBOs)* ao fazer o *rendering* dos modelos. Para além disso, são introduzidas duas novas transformações geométricas, rotação dinâmica e translação sobre uma curva *Catmull-Rom*, ambas sujeitas à variação do tempo. Para esta última, devem ser fornecidos um mínimo de 4 pontos de controlo para definição da curva. Por último, o *generator* devem ainda ser capaz de gerar uma nova primitiva, baseada em *patches* de *Bézier*, recebendo como *input* o nível de tesselação e um ficheiro que define o número de *patches* e os pontos de controlo para um deles. Como *output*, é produzido um ficheiro contendo os triângulos necessários para o desenho, semelhante às restantes primitivas.

0.2 Decisões e abordagens escolhidas

0.2.1 Estruturas de dados

Uma vez que se pretende adicionar duas novas transformações, tal facto obrigou a alterações nas estruturas previamente estabelecidas, bem como a formulação de novas estruturas.

Estrutura *Transform*

A estrutura que suportava as transformações revelou-se insatisfatória de modo a corresponder aos problemas que surgiram nesta fase. Dessa forma, optamos por tornar a mesma numa classe abstrata, representado uma transformação de forma genérica, pelo que cada subclasse desta representará uma transformação em concreto.

```
class Transform {

    public:
        Transform();
        virtual TransformType get_type() const = 0;
};
```

Figura 1: Estrutura *Transform*

Cada classe passa agora a definir as suas próprias variáveis de instância de acordo com a transformação em questão necessita. De notar o método *get_type*, assinalado como *virtual*, que torna a classe, de facto, abstrata. O tipo que este retorna é o que permite distinguir como tratar cada instância desta classe aquando do momento de aplicar as transformações.

```
class StaticRotate : public Transform {

    public:
        angle_t angle;
        CartPoint3d point;

        StaticRotate(angle_t angle, const CartPoint3d &point);
        TransformType get_type() const;
};

class DynamicRotate : public Transform {

    public:
        unsigned time;
        CartPoint3d point;

        DynamicRotate(unsigned time, const CartPoint3d &point);
        TransformType get_type() const;
};
```

Figura 2: Estruturas *StaticRotate* e *DynamicRotate*

As classes da figura 2 são, em tudo, muito semelhantes à exceção dos campos *angle* e *time*, uma vez que a rotação dinâmica é em função do tempo, isto é, *time* indica em quantos segundos deverá ser completada uma rotação, ao passo que a rotação estática é expressa apenas em função do ângulo de revolução.

```
class Scale : public Transform {

    public:
        CartPoint3d point;

        Scale(const CartPoint3d &point);
        TransformType get_type() const;
};
```

Figura 3: Estrutura *Scale*

A classe *Scale* define unicamente o ponto de escala nos três eixos, sendo, por isso, inalterada em relação às fases anteriores.

```
class StaticTranslate : public Transform {

    public:
        CartPoint3d point;

        StaticTranslate(const CartPoint3d &point);
        TransformType get_type() const;
};

class DynamicTranslate : public Transform {

    public:
        unsigned time;
        bool align;
        std::vector<CartPoint3d>* points;

        DynamicTranslate(
            unsigned time, bool align,
            std::unique_ptr<std::vector<CartPoint3d>> &points
        );

        ~DynamicTranslate(); //destrutor to free points

        TransformType get_type() const;
};
```

Figura 4: Estruturas *StaticTranslate* e *DynamicTranslate*

A translação estática apenas define o ponto que representa as coordenadas da translação. No caso da translação dinâmica, define-se o tempo para percorrer a curva *Catmull-Rom* definida em função do vetor de pontos. O campo *align* indica se o modelo se deve alinhar com a curva durante o seu percurso pela mesma. Nota ainda para o destrutor aqui declarado, uma vez que *points* é um apontador para um vetor alocado dinamicamente.

Estrutura *Matrix*

A definição de uma estrutura para representar uma matriz demonstrou-se útil na fase de *rendering*, nomeadamente para as transformações dinâmicas, uma vez que a computação das curvas *Catmull-Rom* se baseia em operações algébricas sobre matrizes. A matriz é implementada usando um *std::array* aninhado dentro de outro. O facto de os *std::array* terem tamanho fixo implica o uso de *templates*, ou seja, programação genérica de modo a que se possam instanciar matrizes de dimensões arbitrárias.

```cpp
template<class T, size_t rows, size_t columns>
struct Matrix {

    std::array<std::array<T, columns>, rows> m;

    constexpr std::array<T, columns>& operator[](size_t i){
        return this->m[i];
    }
};
```

Figura 5: Estrutura *Matrix*

Tal característica da *Matrix* é muito vantajosa para garantir a correção do código, uma vez que funções como aquela cuja a assinatura se apresenta na figura 6 apenas aceitam uma combinação válida de matrizes para o cálculo do seu produto, ou seja, se a matriz $m1$ tem dimensão $l1 \times c1$ e se $m2$ tem $l2 \times c2$, só pode ser computada a multiplicação das mesmas se $c1 = l2$ e se a matriz resultado res tiver dimensão $l1 \times c2$.

```cpp
template<size_t left_size, size_t inner_size, size_t right_size>
static void mult_matrixes(
    Matrix<double, left_size, inner_size> &m1,
    Matrix<double, inner_size, right_size> &m2,
    Matrix<double, left_size, right_size> &res
){
```

Figura 6: Multiplicação de matrizes

Estrutura VBO

Por forma a dar azo ao cumprimento aos requisitos propostos para esta, requer-se que se desenhem os modelos usando *VBOs*, em vez do modo imediato, previamente utilizado. Posto isto, tomou-se a decisão de encapsular a lógica necessária para os *VBOs*, implementando uma classe para esse feito, usando o *singleton design pattern* (isto é, uma única instância da classe), uma vez que as funções de inicialização do *glew* apenas devem ser chamadas **uma única vez**.

```
class VBO{

    private:
        static VBO* singleton;
        std::vector<unsigned> buffers;

        /**
         * For each model, map a pair containing its index and corresponding size,
         * i.e. the number of points to be drawn
         */
        std::map<std::string, std::pair<unsigned, size_t>> model_index_mappings;

        VBO(const std::set<std::string> &models);


    public:
        static VBO* get_instance(const std::set<std::string> &models);
        bool render(const std::string &model) const;
};
```

Figura 7: Estrutura *VBO*

Para cada modelo distinto usa-se um *buffer*. Internamente, a classe possui um *std::map* de modo a fazer corresponder a cada modelo o seu índice - correspondente ao *buffer* adequado - e o número de pontos, necessário para o desenho do modelo em questão. A função *render* recebe como parâmetro o nome do modelo a desenhar.

Primitivas

No que toca à outra aplicação desenvolvida em paralelo, foi introduzido uma nova primitiva, baseada em superfícies de *Bézier*. Inicialmente é feito o *parsing* do ficheiro que contém a informação relativa aos pontos de controlo para cada superfície, com as devidas verificações (nomeadamente garantir que cada *patch* especifica 16 pontos para a interpolação).

Segue-se o cálculo dos vários pontos que formam os triângulos da superfície. O nível de tesselação indica o número de divisões que deverão ser obtidas para a superfície. É, por outras palavras, o nível de detalhe do modelo. Ora, para n divisões são necessários $n+1$ pontos. Daí resulta que obtemos uma matriz auxiliar $(n+1) \times (n+1)$. Esta matriz representa, no fundo, a grelha dos pontos da superfície a gerar, pelo que basta escreve-los pelo ordem adequada para ficheiro de modo a obter os triângulos. Cada cada valor $p(u, v)$ da mesma é calculado a partir de quatro pontos auxiliares, obtidos a partir da fórmula para o cálculo de um ponto numa curva de *Bézier* cúbica, usando como coeficientes binomiais o vetor $[t^3, t^2(1-t), t(1-t)^2, (1-t)^3]$, obtido em função de u. Como se tratam de curvas de nível 3, são necessários quatro pontos em cada uma das quatro curvas, totalizando 16, como anteriormente referido. Estes pontos auxiliares são então aplicados à mesma fórmula para cálculo de um numa curva, desta vez com o vetor dos coeficientes em função de v. O algoritmo tem, por isso, complexidade $O(n^2)$.

0.2.2 Correções e otimizações

Não houve alterações significativas face ao que já havia sido implementado, à exceção do facto do *engine* suportar os dois modos na fase de desenho, isto é, com ou sem *VBOs* ativados. Para tal, basta especificar como argumento (opcionalmente) *y* para sim ou *n* caso contrário. Por omissão, são usados *VBOs*. O *engine* aceita ainda um outro argumento opcional, que define o nível de tesselação da curva *Catmull-Rom*, caso se aplique. Por omissão, assume-se um valor de 100.

0.3 Testes

0.3.1 teste_3_1

Captamos dois instantes do *teste_3_1* para demonstrar o movimento com a passagem do tempo.

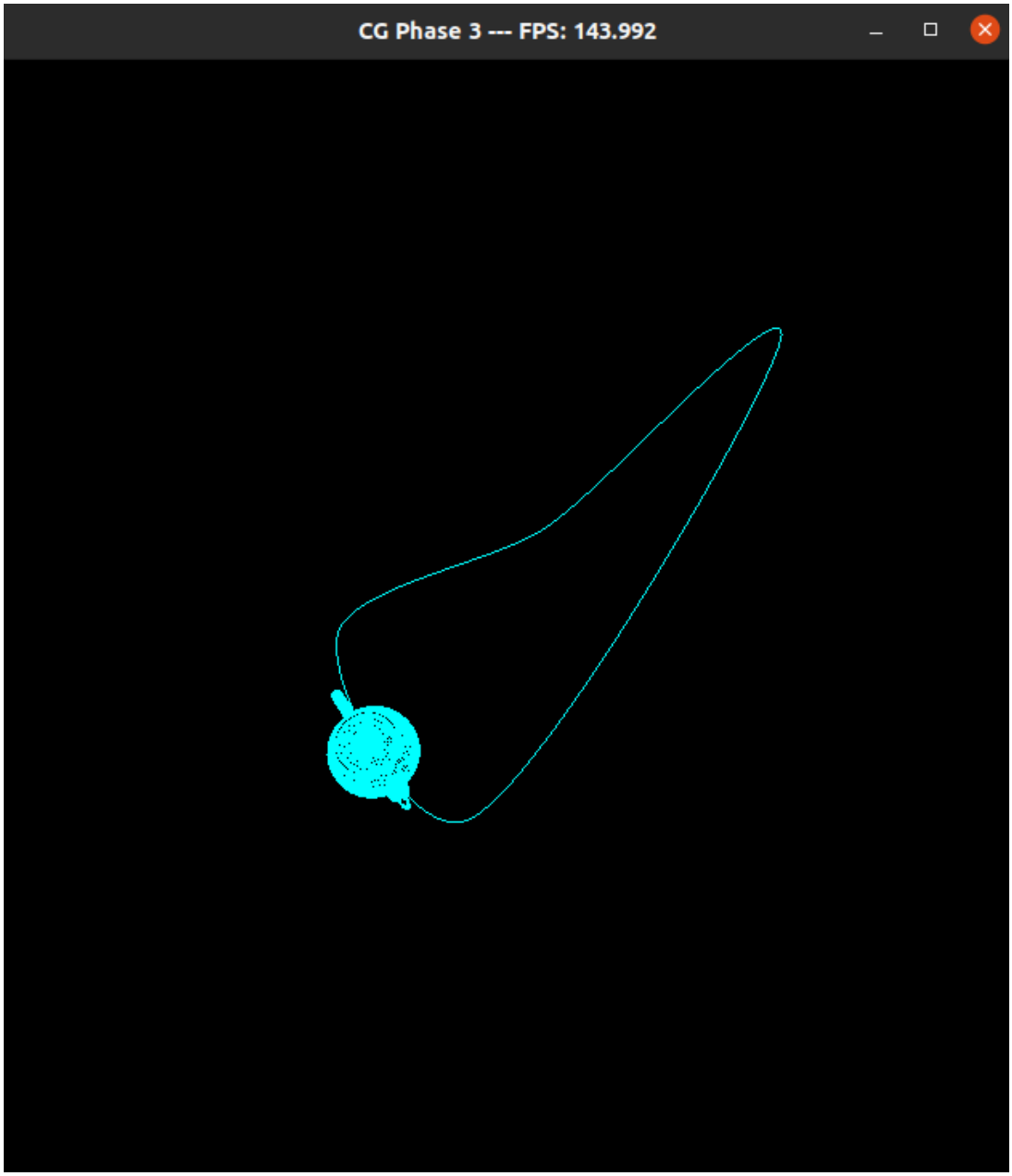


Figura 8: 1º instante

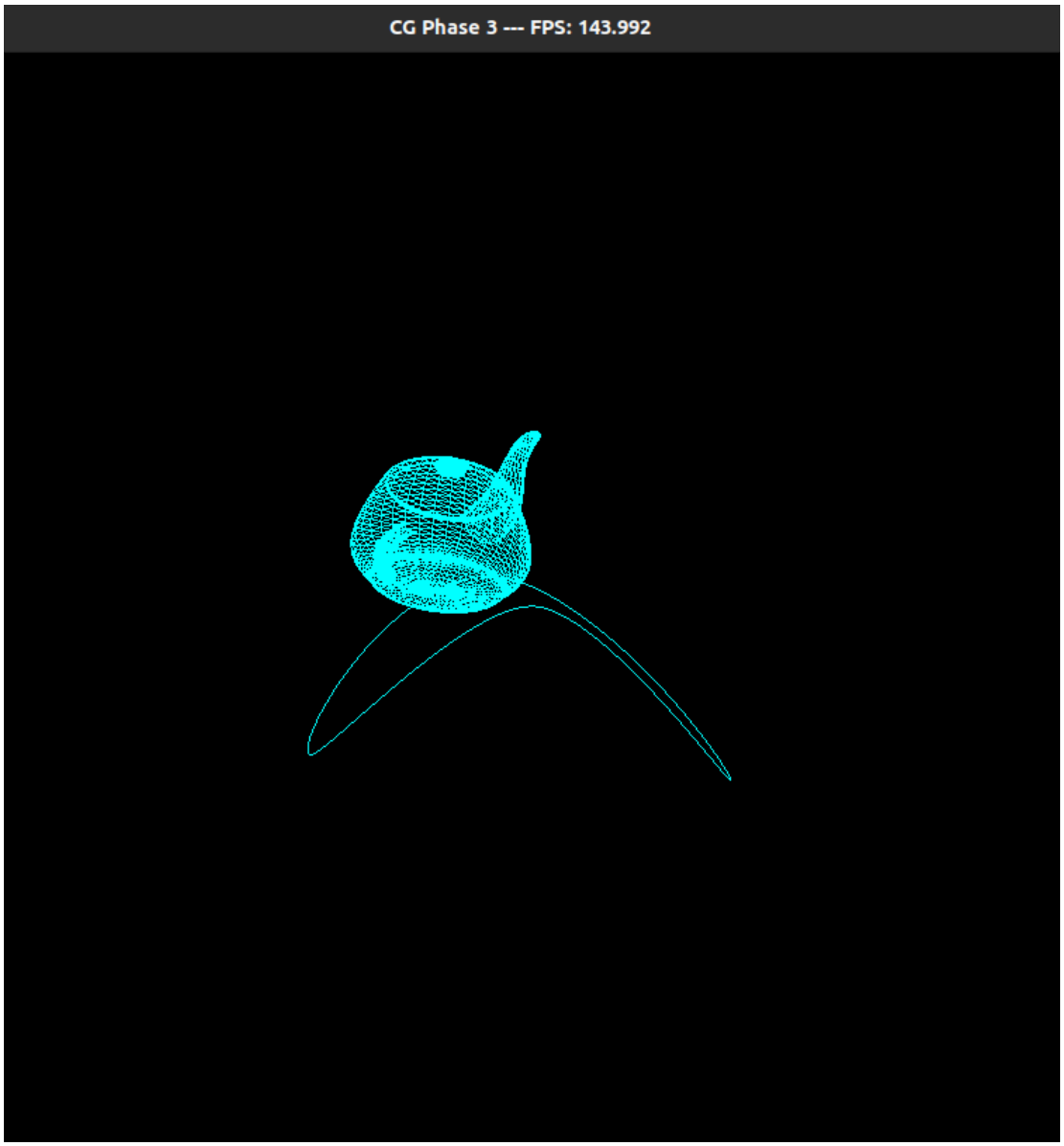


Figura 9: 2º instante

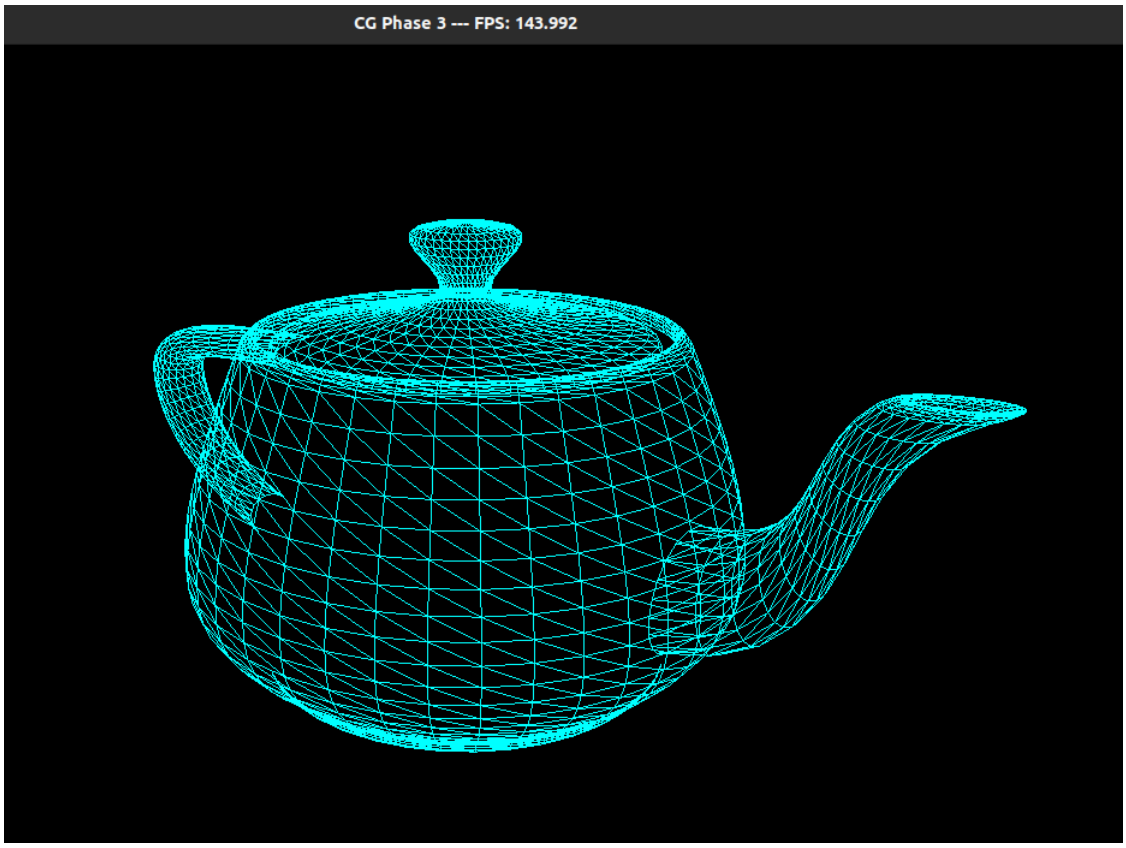
FPS Table

Na tabela seguinte, estão os valores FPS (*frames per second*) obtidos na execução do *teste_3_1* com várias valores de tesselação e com estados diferentes (*no* ou *yes*) de utilização de *VBOs (Vertex Buffer Object)*. A máquina que foi utilizada tem 16GB RAM, processador Intel Core i5 e placa gráfica NVIDIA GeForce RTX 3060. O código foi compilado com otimização nível 2 (*flag -O2*), C++ versão 17.

Tesselação	Máximo	Mínimo	Média(de 15)	VBO
8	146.414	143.849	144.164	no
8	146.414	143.849	144.202	yes
16	146.707	143.856	144.184	no
16	146.971	143.856	144.201	yes
32	146.414	143.426	144.155	no
32	146.123	143.849	144.144	yes
64	142.715	143.878	138.502	no
64	146.56	143.849	144.173	yes
128	27.4241	26.1375	26.6804	no
128	85.9684	80.9191	82.7245	yes

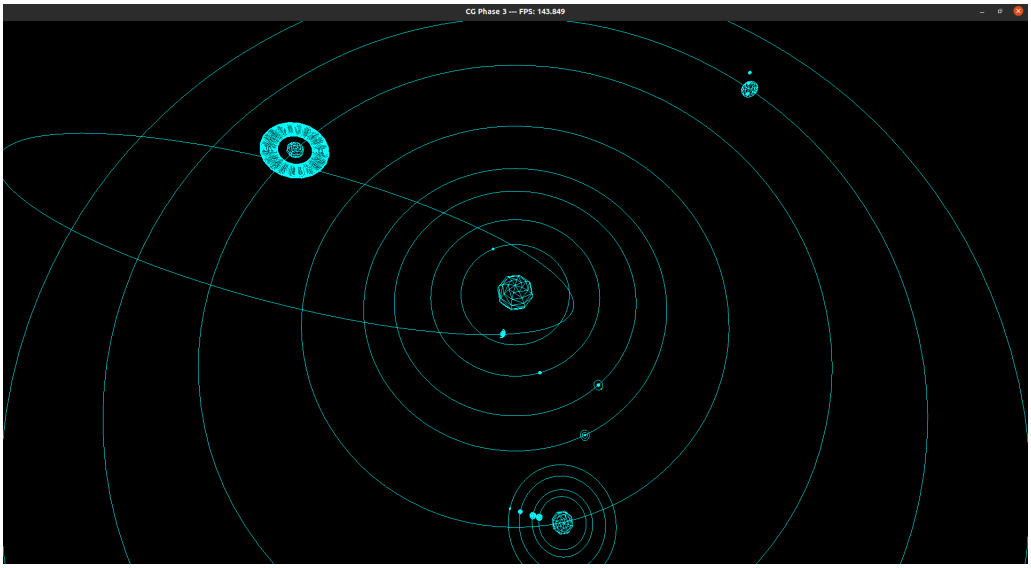
Infelizmente, devido a uma restrição do número máximo de FPS da máquina onde foi testado, não foi possível obter dados muito revelantes em valores de tesselação baixos, mas foi possível observar a diferença entre a utilização e não utilização de *VBOs* quando o valor de tesselação é 128, o que permite concluir que o uso de *VBOs* torna o *rendering* mais eficiente.

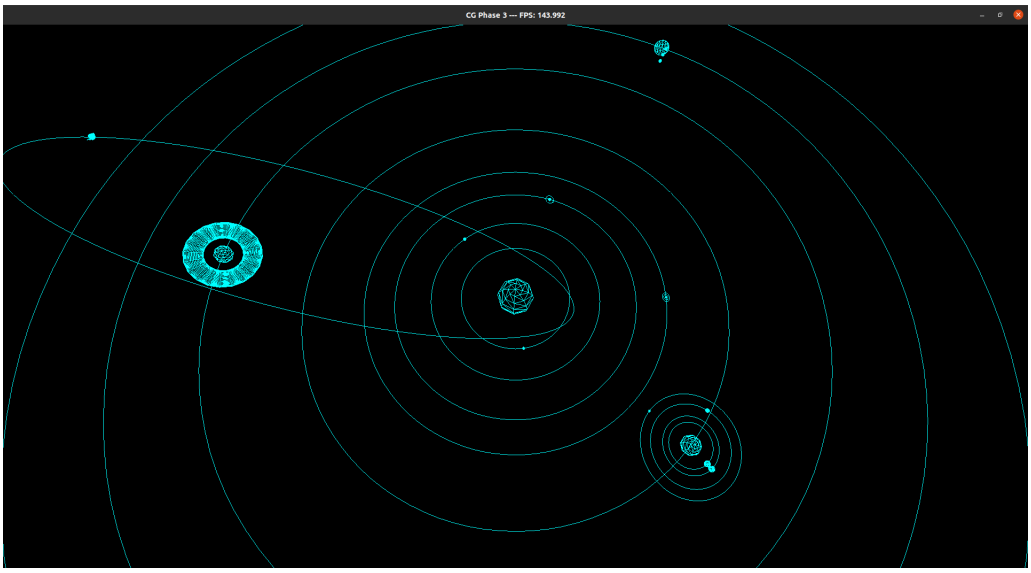
0.3.2 teste_3_2



0.3.3 sistema_solar

Como existe movimento no nosso demo, optamos por captar dois instantes do mesmo para o demonstrar.





0.4 Conclusão

Concluída esta terceira fase do trabalho prático, consideramos que atingimos os objetivos exigidos para mesma de forma satisfatória e elegante. Demos, assim, mais um passo em direção à conclusão do projeto, com uma base para a última fase consolidada.