Imatge Sintètica

Ray Tracing for Realistic Image Synthesis

Ricardo Marques
(ricardo.marques@upf.edu)

Group de Tecnologies Interactives (GTI)
Departament de Tecnologies de la Informació i les Comunicacions (DTIC)
Universitat Pompeu Fabra (UPF)

Edifici Tànger - Office 55.106

Framework - Vector3D, Matrix3D and Film

2017/2018

Section 1

Vector3D

Vector3D - Definition

```
struct Vector3D
{
    (...)

    // Structure data
    double x, y, z;
};
```

- Used to represent different entities composed of a 3-tuple
  - RGB color, vectors, normals, points
  - We can access each element (x, y, z) of a variable **v** of type *Vector3D* using the C++ *member selection operator* "."
    - Example **v** . x
    - x, y and z are of type *double*
- **Important Note**: it is up to the programmer to distinguish between points, colors, vectors and normals!

Vector3D - Constructors

- *Vector3D(...)* is called a constructor of *Vector3D*
- A *Vector3D* can be *constructed* in different ways:

```
// Constructors
Vector3D();   // Default constructor
Vector3D(double a);
Vector3D(double x, double y, double z);
Vector3D(const Vector3D &v);
```

- By default, a *Vector3D* has value $(0, 0, 0)$
- When constructed from another *Vector3D*, the new variable of type *Vector3D* will take the value of original variable

Vector3D - Operators

- The implementation of the structure *Vector3D* offers a number of operations over vectors

```
    Vector3D operator+(const Vector3D &v) const;
    Vector3D operator-(const Vector3D &v) const;
    Vector3D operator*(const double a) const;
    Vector3D operator/(const double a) const;
    Vector3D operator-() const;

    Vector3D& operator+=(const Vector3D &v);
    Vector3D& operator-=(const Vector3D &v);
    Vector3D& operator*=(const double a);
    Vector3D& operator/=(const double a);
```

- A Vector3D can also be written to the standard output

```
std::ostream& operator<<(std::ostream& out,
                         const Vector3D &v);
```

Vector3D - Others

- The implementation of the structure *Vector3D* offers a number of operations over vectors

```
    // Member functions
    double length()        const;
    double lengthSq()      const;
    Vector3D normalized()  const;
```

- You can also compute the dot product and the cross product between two variables of type *Vector3D*

```
double dot(const Vector3D &v1, const Vector3D &v2);
Vector3D cross(const Vector3D &v1, const Vector3D &v2);
```

- **Attention**: these operations might not be appropriate for some entities represented using a Vector3D
  - Example: computing the length (magnitude) of a point

Section 2

Matrix4x4

Matrix4x4 - Definition

- Used to represent transformation matrices in homogeneous coordinates (4x4)

```
struct Matrix4x4
{
    (...)

     // Structure data
     double data[4][4];
};
```

- Can be written to the standard output

```
// Stream insertion operator
ostream& operator<<(ostream &out, const Matrix4x4& m);
```

Matrix4x4 - Constructors

- A *Matrix4x4* can be *constructed* in different ways:

```
// Constructors
Matrix4x4(); // Default Constructor

Matrix4x4(double data_[4][4]);

Matrix4x4(
    double a00, double a01, double a02, double a03,
    double a10, double a11, double a12, double a13,
    double a20, double a21, double a22, double a23,
    double a30, double a31, double a32, double a33
    );
```

- By default, a matrix is constructed as an *identity matrix*

Matrix4x4 - Operators

- The implementation of the structure *Matrix4x4* offers a number of operations over matrices

```
// Member operators overload
Matrix4x4 operator+(const Matrix4x4 &m) const;
Matrix4x4 operator-(const Matrix4x4 &m) const;
Matrix4x4 operator*(const Matrix4x4 &m) const;
Matrix4x4 operator*(const double      a) const;
```

- Common matrix manipulations

```
bool inverse(Matrix4x4 &target) const;
void setToZeros();
void transpose(Matrix4x4 &target) const;
```

Matrix4x4 - Others

Matrix4x4 offers static methods to construct commonly used transformation matrices

```
static Matrix4x4 translate(const Vector3D &delta);
static Matrix4x4 scale(const Vector3D &scaleVector);
static Matrix4x4 rotate(const double angleInRad,
                        const Vector3D &axis);
```

- And to apply the transformation contained by the matrix

```
// Member functions (Transformations)
Vector3D transformVector(const Vector3D &v) const;
Vector3D transformPoint(const Vector3D &p) const;
Ray      transformRay(const Ray &r) const;
```

- **Note:** there is no “transformNormal()”. You can implement it!

Section 3

Film

Film - Definition

- Used to hold the image data

```
// Member functions (Transformations)
class Film
{
public:
        (...)

private:
    // Image size
    size_t width;
    size_t height;

    // Pointer to image data (bidimensional!)
    Vector3D **data;
};
```

Film - Constructor

- The *Film* class has a single constructor (which allocates the memory space for the image according to its resolution)

```
// Constructor(s)
Film(size_t width_, size_t height_);
Film() = delete;
```

- And a destructor explicitly defined (which de-allocates the memory space used for the image)

```
// Destructor
~Film();
```

Film - Methods

- The implementation of the structure *Film* offers a number of operations over images

```
// Getters
size_t getWidth() const;
size_t getHeight() const;
Vector3D getPixelValue(size_t w, size_t h) const;

// Setters
void setPixelValue(size_t w, size_t h,
Vector3D &value);

// Other functions
int save();
void clearData();
```