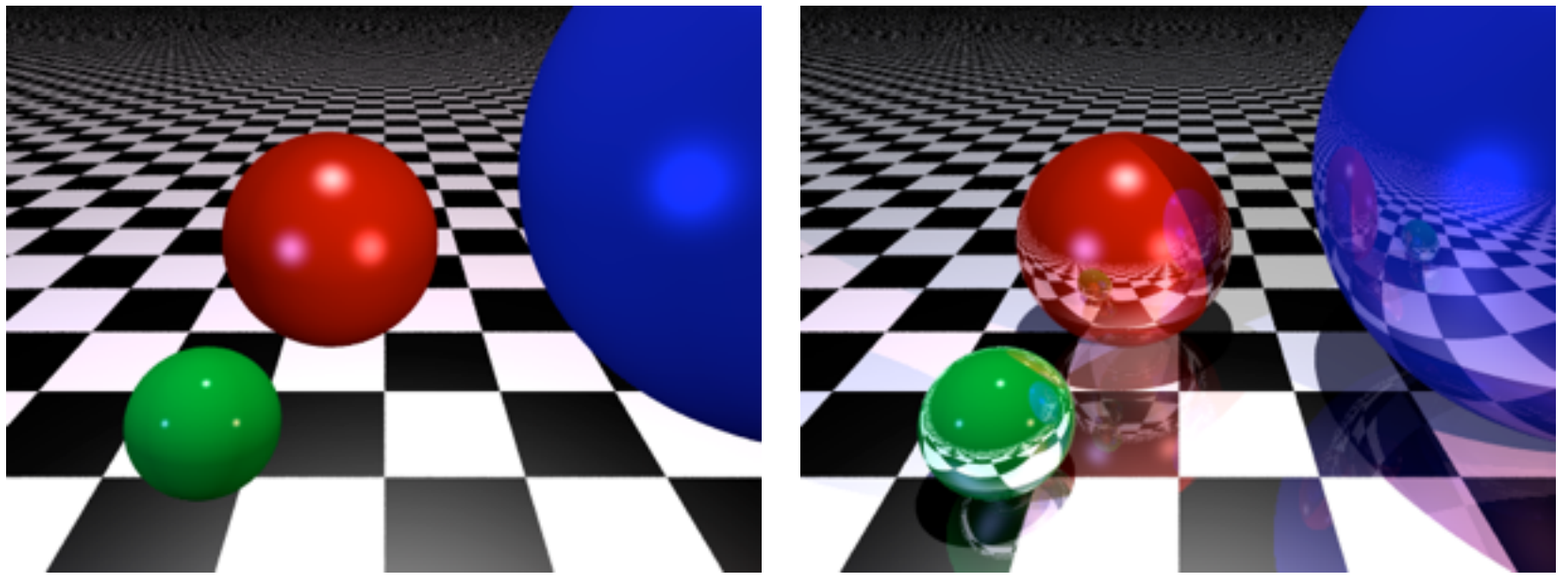
Introduction to Computer Graphics

Implementing a Ray Tracer



Prof. Dr. Mario Botsch

Computer Graphics & Geometry Processing

Universität Bielefeld

CITEC

Outline

- **C++ Crash Course**
- RayTracer Design
- Performance optimization
- Multicore parallelization

What is C++?

- C++ is a multi-paradigm general purpose programming language
- C++ emerged as an extension to C providing features such as OOP, exceptions and generic programming

Basic Data Types

- C++ has 7 basic data types:

Type	Description
`void`	no associated type
`int`	integer numbers
`float`	floaring point numbers
`double`	double precision floats
`char`	characters
`bool`	boolean, true or false
`wchar_t`	wide character

- Additional modifiers:
  - `signed`, `unsigned`, `short`, `long`

Control Structures

```
for (int i = 0; i < 10; i++)
{
    // do something useful
}
```

```
while (i < 10)
{
    // do something useful
    i++;
}
```

```
do
{
    // do something useful
}
while (!finished);
```

```
switch (foo)
{
  case 1:
   func1();
   break;

  case 2:
   func2();
   break;

  default:
   default_func();
   break;
}
```

Pointers & References

- Pointers are variables that “point” to a specific address in memory

```
int a;     // variable a
int* ptr; // pointer ptr

&a         // address of a;
*ptr       // data of pointer
```

```
int a = 5;
int* ptr = &a;
a = 11;
int b = *ptr;
*ptr = 27;

int a2 = 3;
ptr = &a2;
```

Arrays

- Collections of elements of the same type

```
int my_vector[3];

for (int i = 0; i < 3; i++)
{
  my_vector[i] = i;
}
```

```
int my_vector[3] = {0, 1, 2};

my_vector[0] = 11;
```

- Pointers reviewed

```
int* ptr = &my_vector[1];
int a = *ptr;
int b = ptr[1];
int c = ptr[2];
```

Classes

```
// vec2.h
class vec2
{
public:
  vec2(double _x, double _y);
  void set(double _x, double _y);

protected:
  double x_, y_;

private:
  // ...
};
```

```
// vec2.cpp
#include “vec2.h”

vec2::vec2(double _x, double _y)
{
  x_ = _x;
  y_ = _y;
}

void vec2::set(double _x, double _y)
{
  x_ = _x;
  y_ = _y;
}
```

Pointers reviewed

```
// main.cpp
vec2 v(1.5, 2.0);
v.set(5.0, 8.0);

vec2* vec_ptr = &v;
vec_ptr->set(5.0, 8.0);
```

Initialization

```
// vec2.h
class vec2
{
public:
  vec2(double _x, double _y);
  void set(double _x = 1.0,
           double _y = 0.0);

protected:
  double x_, y_;

private:
  // ...
};
```

Default arguments

```
// vec2.cpp
vec2::vec2(double _x, double _y)
  : x_(_x), y_(_y) { }

void vec2::set(double _x, double _y)
{
  x_ = _x;
  y_ = _y;
}
```

Initialization lists

```
// main.cpp
vec2 vec(1.5, 2.0);
vec.set();

vec.set(2.0f);
```

C++ Memory Management

- Manual memory management, no built-in garbage collection
- Local variables are allocated and destroyed automatically
- Variables created on the heap have to be deleted manually (`new`/`delete` or `new[]`/`delete[]`)

Destruction

```h
// vec2.h
class vec2
{
public:
  vec2(double _x, double _y);

  ~vec2();

  // ...
};
```

```cpp
// vec2.cpp
vec2::~vec2() {
  std::cout << "I just got destructed" << std::endl;
}

// ...
```

```cpp
// main.cpp
{
  vec2 vec(1.5, 2.0);
  // ...

} // "I just got destructed"
```

C++ Memory Management

```
{
  vec2 vec(1.5, 2.0);
  // ...
} // "I just got destructed"
```

```
{
  vec2* vec = new vec2(1.5, 2.0);
  // ...
  delete vec; // "I just got destructed"
  // ...
  vec->set(1.0f, 2.0f);
}
```

“memory leak”

```
{
  vec2* vec = new vec2(1.5, 2.0);
  // ...
}
```

behavior “undefined”

```
{
  vec2* vecs = new vec2[10];
  // ...
  delete[] vecs; // 10x "I just got destructed"
  // ...
}
```

The *const* Qualifier

- Declaring variables as const

```
const int foo = 42;

foo = 5;
```

- Declaring parameters as const

```
void foo(const vec2& _b, vec2& _x)
{
   _x.set(1.0, 2.0);

   _b.set(3.0, 4.0);
}
```

Constant member-functions

```
// vec2.h
class vec2
{
public:
  // ...
  double dot(const vec2& _v) const
  {
    return x_*_v.x_ + y_*_v.y_;
  }

  void set(double _x, double _y) const
  {
    x_ = _x;
    y_ = _x;
  }
};
```

“read-only”

“write-function”

Operator overloading

```
// vec2.h
class vec2
{
public:
  // ...
  double operator+(const vec2& _v)
  {
    return vec2(x_ + _v.x_, y_ + _v.y_);
  }

  // ...
};
```

```
// main.cpp
vec2 v1(1.5, 2.0);
vec2 v2(4.0, 2.3);

vec2 sum = v1+v2;
```

Namespaces

- Namespaces allow to group entities under a common name, similar to Java packages

```
//somewhere in <iostream>
namespace std {
   // ...
   // definition of cout
   // definition of endl
   // ...
}
```

```
#include <iostream>

int main(void) {
    std::cout << "asdf!" << std::endl;
}
```

```
#include <iostream>

//using namespace std;
using std::cout;
using std::endl;

int main(void) {
    cout << "asdf!" << endl;
}
```

Basic Templates

- C++ templates allow for generic programming, i.e. writing algorithms without explicitly specifying types in the first place

```
template <class T>
T max(T x, T y)
{
    return y > x ? y : x;
}

// ...

int    a1 = max(3, 7);
double a2 = max(3.2, 7.4);
```

The Standard Template Library

- Provides containers, iterators, and commonly used algorithms
- Example: The `std::vector<>` template provides a dynamic array

```
{
  std::vector<vec2> vectors;

  vectors.push_back(vec2(0.8, 0.2)); // append
  vectors.push_back(vec2(0.9, 0.4));

  vectors[1].set(3.0, 4.0);          // random access

} // all elements get destructed & all memory is freed at this point
```

The Standard Template Library

- Provides containers, iterators, and commonly used algorithms
- Containers:
  - dynamic array: `std::vector<T>`
  - linked list: `std::list<T>`
  - stack (LIFO): `std::stack<T>`
  - queue (FIFO): `std::queue<T>`
  - priority queue: `std::priority_queue<T>`

The Standard Template Library

- Iterators provide a *generic* way to enumerate all elements of a container

```
#include <vector>

std::vector<int>  my_array;
my_array.push_back(42);

int sum(0);
for (std::vector<int>::iterator it=my_array.begin(); it!=my_array.end(); ++it)
{
    sum += *it;
}
```

```
#include <list>

std::list<int>  my_list;
my_list.push_back(42);

int sum(0);
for (std::list<int>::iterator it = my_array.begin(); it != my_array.end(); ++it)
{
    sum += *it;
}
```

C++11

- C++11 provides several cool new features
  - auto keyword and range-based for loops

```
// old C style (only works for arrays)
for (int i=0; i<my_array.size(); ++i)
   sum += my_array[i];
```

```
// "old" C++ iterators (works for all STL containers)
for (std::vector<int>::iterator it=my_array.begin(); it!=my_array.end(); ++it)
   sum += *it;
```

```
// using the "auto" keyword the compiler determines the type automagically
for (auto it=my_array.begin(); it!=my_array.end(); ++it)
   sum += *it;
```

```
// range-based for loops simplify code even more
for (auto i : my_array)
   sum += i;
```

C++11

- C++11 provides several cool new features
  - pointers vs. shared pointers

```
{
    // setup scene with a sphere and a plane
    typedef Object* ObjectPtr;
    std::vector<ObjectPtr> objects;
    objects.push_back( new Sphere(...) );
    objects.push_back( new Plane(...) );

    // raytrace scene...
    compute_image();

    // object have been allocated with new so they have to be deleted
    for (auto object : objects)
    {
        delete object;
    }
}
```

C++11

- C++11 provides several cool new features
  - pointers vs. shared pointers

```
{
    // setup scene with a sphere and a plane
    typedef shared_ptr<Object> ObjectPtr;
    std::vector<ObjectPtr> objects;
    objects.push_back( ObjectPtr(new Sphere(...)) );
    objects.push_back( ObjectPtr(new Plane(...)) );

    // raytrace scene...
    compute_image();

    // nice: objects are deleted automaticall when vector is destroyed
    // not so nice: shared pointers are slower than standard pointers
}
```

WER WIRD

Computergrafiker

WER WIRD

Questionnaire

For two orthogonal vectors it holds:

(A) $\mathbf{a}^T\mathbf{b}=0$ and $\mathbf{a}\times\mathbf{b}=\mathbf{0}$

(B) $\mathbf{a}^T\mathbf{b}\neq 0$ and $\mathbf{a}\times\mathbf{b}\neq\mathbf{0}$

(C) $\mathbf{a}^T\mathbf{b}\neq 0$ and $\mathbf{a}\times\mathbf{b}=\mathbf{0}$

(D) $\mathbf{a}^T\mathbf{b}=0$ and $\mathbf{a}\times\mathbf{b}\neq\mathbf{0}$

Barycentric Coordinates

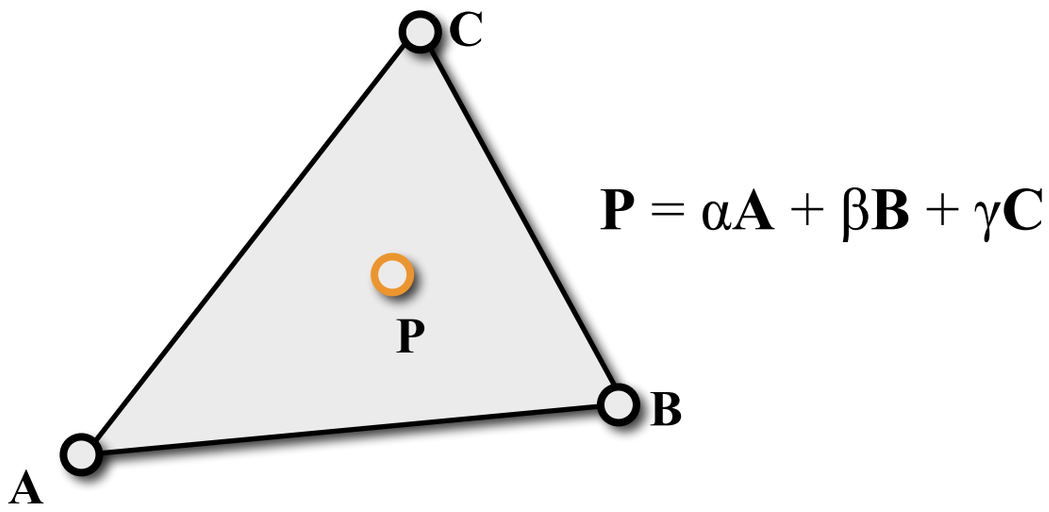
When is $\mathbf{P} = \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C}$ inside triangle (**A**,**B**,**C**)?

(A) If $\alpha + \beta + \gamma = 1$

(B) If $\alpha < \beta < \gamma$

(C) If $\alpha, \beta, \gamma < 1$

(D) If $\alpha, \beta, \gamma > 0$



Questionnaire

What is the relation between the number of vertices V and the number of faces F in a triangle mesh?

(A) $V \approx 2F$

(B) $F \approx 2V$

(C) $V \approx 3F$

(D) $F \approx 3V$

Questionnaire

Which color model covers the largest amount of human-visible colors?

(A) RGB

(B) HSV

(C) CIE

(D) CMYK

Lighting

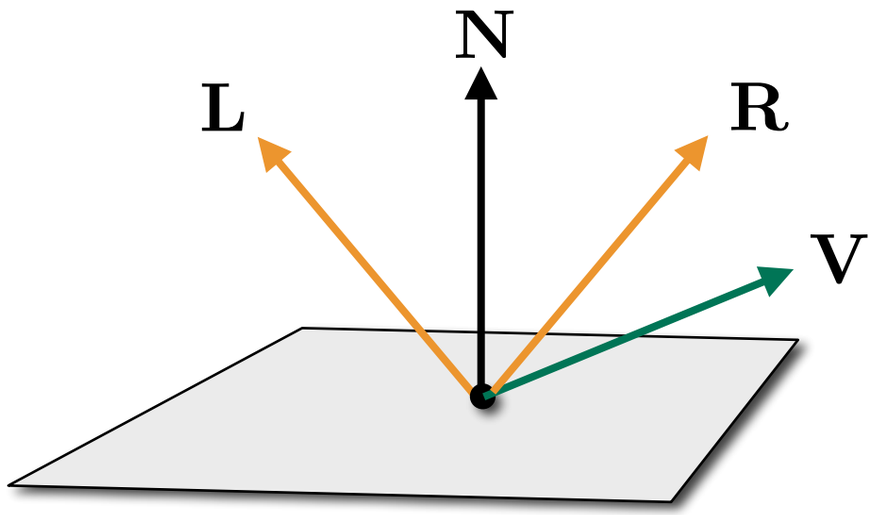
What is the diffuse component of Phong lighting?

(A) $I_l\, k_d\, (\mathbf{R}\cdot\mathbf{V})$

(B) $I_l\, k_d\, (\mathbf{N}\times\mathbf{L})$

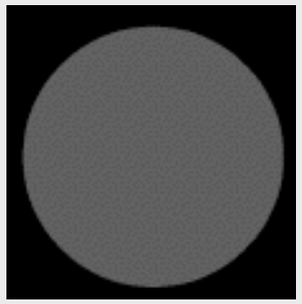
(C) $I_l\, k_d\, (\mathbf{N}\cdot\mathbf{L})$

(D) $I_l\, k_d\, (\mathbf{R}\times\mathbf{V})$

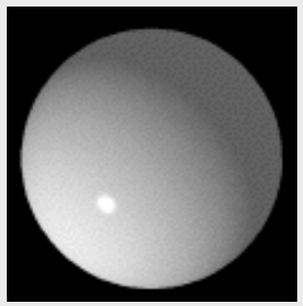


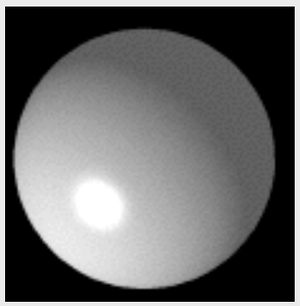
Lighting

ambient	diffuse	specular	shininess
0,3	0	0	0
0,3	0,6	0	0
0,3	0,6	1	20
0,3	0,6	1	200

(A) 

(B) 

(C) 

(D) 

Outline

- C++ Crash Course
- **RayTracer Design**
- Performance optimization
- Multicore parallelization

3D Vectors

```
class vec3
{
private:

    double data_[3];

public:

    /// construct with x,y,z values
    vec3(double _x=0.0, double _y=0.0, double _z=0.0);

    /// access elements by index
    double& operator[](unsigned int _i);

    /// access elements by index
    double operator[](unsigned int _i) const;

    /// vector += vector
    vec3& operator+=(vec3 v);

    // ...
};


/// vector addition
vec3 operator+(vec3 v0, vec3 v1);

/// dot product
double dot(vec3 v0, vec3 v1);
```

Material & Light

```
/// material stores the parameters required for Phong lighting
struct Material
{
    vec3   ambient;
    vec3   diffuse;
    vec3   specular;
    double shininess;
    double mirror;
};

/// light is specified by position and color
struct Light
{
    vec3 position;
    vec3 color;
};
```

Ray

```
class Ray
{
public:

    /// constructor
    Ray(const vec3& _o=vec3(0,0,0), const vec3& _d=vec3(0,0,-1))
    : origin(_o), direction(normalize(_d)) {}


    // evaluate point on ray
    vec3 operator()(double _t) const
    {
        return origin + _t*direction;
    }


public:

    vec3 origin;
    vec3 direction;
};
```

Base Class for Geometric Objects

```
struct Object
{
public:

    /// constructor
    Object() {}

    /// destructor (has to be virtual!)
    virtual ~Object() {}

    /// intersect object with _ray, return intersection data.
    /// function has to be overloaded in derived classes
    virtual bool intersect(const Ray&  _ray,
                           vec3&       _intersection_point,
                           vec3&       _intersection_normal,
                           double&     _intersection_t) const = 0;

    /// material: ambient, diffuse, specular, shininess, mirror
    Material material;
};
```

Derived Classes for Geometric Objects

```
class Plane : public Object
{
public:

    /// constructor
    Plane(const vec3& _center=vec3(0,0,0), const vec3& _normal=vec3(0,1,0));

    /// compute plane-ray intersection
    virtual bool intersect(const Ray&  _ray,
                           vec3&       _intersection_point,
                           vec3&       _intersection_normal,
                           double&     _intersection_t) const;

public:

    vec3 center, normal;
};
```

```
/// read plane from input stream
inline std::istream& operator>>(std::istream& is, Plane& p)
{
    is >> p.center >> p.normal >> p.material;
    return is;
}
```

Camera

```
class Camera
{
public:

    Camera(const vec3&   _eye,
           const vec3&   _center,
           const vec3&   _up,
           double        _fovy,
           unsigned int  _width,
           unsigned int  _height);

   Ray primary_ray(unsigned int _x, unsigned int _y) const;


public:

    vec3    eye, center, up;  // eye point, look-at-point, up-vector
    double  fovy;             // opening angle (field of view) in y-direction
    int     width, height;    // image resolution
};
```

Main Loop

```
// the main ray tracing loop
for (unsigned int y=0; y<camera.height; ++y)
{
    for (unsigned int x=0; x<camera.width; ++x)
    {
        // generate primary ray for pixel (x,y)
        Ray ray = camera.primary_ray(x,y);

        // trace ray and write color to image
        image(x,y) = trace(ray, 0);
    }
}
```

Outline

- **C++ Crash Course ✓**
- **RayTracer Design ✓**
- Performance optimization
- Multicore parallelization

Literature

- Bjarne Stroustrup: ***The C++ Programming Language***, 4th edition, Addison-Wesley, 2013