Computer Vision
and Geometry Lab

Informatik I for D-MAVT

Exercise Session 11

Themen

- Bemerkungen
- Vererbung
- Vorbesprechung Serie 11
- Nachbesprechung Serie 9: Rekursion

Bemerkungen

- Code sollte zumindest ohne Fehler kompilierbar sein!
- Falls es Probleme beim Kompilieren gibt, so sollten diese beschrieben werden
- Formatierung des Codes (Tabulator..?)

Vererbung: Motivation

- Weiterverwenden von Code
- Strukturiertes Design
- Spezialisierung erfolgt in abgeleiteten Klassen
- Ermöglicht strikte Trennung zwischen Interface und Implementierung
  - Interface: wie mit einem Objekt interagiert werden kann
  - Implementierung: wie die durch eine Interaktion hervorgerufene Zustandsänderung effektiv berechnet und gespeichert wird
    - Implementierung ist für Benutzer einer Klasse eigentlich nicht von Interesse!

Vererbung: Grundkonzept

- Basisklasse (auch Parent Class genannt)
- Abgeleitete Klassen (auch Child Class genannt)
  - Erben die Elemente (Attribute und Methoden) der Basisklasse
    - Access Specifier legen Zugriffsrechte in abgeleiteter Klasse fest
  - Zusätzlich neue eigene Elemente möglich
    - Stehen der Basisklasse nicht zur Verfügung

Vererbung: Access Specifier

- protected
  - Abgeleitete Klasse hat Zugriff
- private
  - Abgeleitete Klasse hat keinen Zugriff

```
class A
{
public:
        A(void){}
        ~A(void){}
protected:
        char c;
};
```

```
class A
{
public:
        A(void){}
        ~A(void){}
private:
        char c;
};
```

Vererbung: Access Specifier

```
#include <iostream>
class A
{
public:
        A(void){c = 'a';}
        ~A(void){}
        void print(){std::cout << "A: " << c << std::endl;}
protected:
        char c;
};


class B : public A
{
public:
        B(void){c = 'b';}
        ~B(void){}
        void print(){std::cout << "B: " << c << std::endl;}
};
```

```
#include "A.h"
#include "B.h"

int main()
{
        A a;
        B b;
        a.print();
        b.print();
        return 0;
}
```

A: a

B: b

Basisklasse und abgeleitete Klasse können Funktionen mit identischen Namen und Signaturen enthalten!

Vererbung: Access Specifier

```
#include <iostream>
class A
{
public:
        A(void){c = 'a';}
        ~A(void){}
        void print(){std::cout << "A: " << c << std::endl;}
private:
        char c;
};


class B : public A
{
public:
        B(void){c = 'b';}
        ~B(void){}
        void print(){std::cout << "B: " << c << std::endl;}
};
```

```
#include "A.h"
#include "B.h"

int main()
{
        A a;
        B b;
        a.print();
        b.print();
        return 0;
}
```

Compiler Error

Vererbung: Access Specifier

- Syntax

```
class B: public A
class B: protected A
class B: private A
```

- public: Abgeleitete Klasse übernimmt Access Specifier für geerbte Elemente
- protected: Abgeleitete Klasse setzt public auf protected für geerbte Elemente
- private: Alle geerbten Elemente der abgeleiteten Klasse sind private

Vererbung: Access Specifier

```
class A
{
public:
        A(void){c = 'a';}
        ~A(void){}
        void print(){std::cout << c << std::endl;}
protected:
        char c;
};
```

```
class B : public A
{
public:
        B(void){c = 'b';}
        ~B(void){}
};
```

print() für Objekte der Klasse B ist public

c ist protected

```
class B : protected A
{
public:
        B(void){c = 'b';}
        ~B(void){}
};
```

print() für Objekte der Klasse B ist protected

c ist protected

```
class B : private A
{
public:
        B(void){c = 'b';}
        ~B(void){}
};
```

print() für Objekte der Klasse B ist private

c ist private

Vererbung: Konstruktor und Destruktor

- Destruktor und Konstruktoren werden nicht vererbt
- Der Default-Konstruktor der Basisklasse wird von allen Konstruktoren der abgeleiteten Klasse aufgerufen, falls nichts anderes spezifiziert wurde
- Der Destruktor der Basisklasse wird in der abgeleiteten Klasse aufgerufen, nachdem der Destruktor der abgeleiteten Klasse aufgerufen wurde

Vererbung: Konstruktor und Destruktor

```
class A
{
public:
        A(void){c = 'a';}
        ~A(void){}
protected:
        char c;
};


class B : public A
{
public:
        B(void){c = 'b';}
        ~B(void){}
};
```

Funktioniert

```
class A
{
public:
        A(char c){this->c = c;}
        ~A(void){}
protected:
        char c;
};


class B : public A
{
public:
        B(void){c = 'b';}
        ~B(void){}
};
```

Kompiliert nicht

```
class A
{
public:
        A(char c){this->c = c;}
        ~A(void){}
protected:
        char c;
};


class B : public A
{
public:
        B(void): A('b'){}
        ~B(void){}
};
```

Funktioniert

Vererbung: Konstruktor und Destruktor

```
// constructors and derived classes
#include <iostream>
using namespace std;

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (0);
  son daniel(0);
  return 0;
}
```

Output:

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

http://www.cplusplus.com/doc/tutorial/inheritance/

Vererbung: Assignment Operator

- Falls der Assignment-Operator nicht definiert wird in der abgeleiteten Klasse, so wird der Assignment-Operator der Basisklasse aufgerufen. Nicht geerbte Elemente werden einfach kopiert.

Vererbung: Assignment Operator

```
class A
{
public:
        A(char c){this->c = c;}
        A & operator = (const A & other){
                if (this != &other){
                        c = other.c;
                }
                return *this;
        }
        ~A(void){}
        void print(){std::cout << c << std::endl;}
protected:
        char c;
};
```

```
class B : public A
{
public:
        B(void): A('b'){}
        ~B(void){}
public:
        char c2;
};
```

```
int main()
{
        A a('a');
        B b,b2;
        b.c2 = 'x';
        b2 = b; // b2.c2 == 'x'
        return 0;
}
```

Vererbung: Polymorphismus

- Auf Objekte der abgeleiteten Klasse kann über einen Pointer zur Basisklasse zugegriffen werden
- Verfügen Basisklasse und abgeleitete Klassen über gleichnamige Funktionen mit identischer Signatur, so wird die „passende“ Methode zur Laufzeit bestimmt
- Keyword virtual

Vererbung: Polymorphismus

```
#include <iostream>
class A
{
public:
        A(void){c = 'a';}
        ~A(void){std::cout << "destructor A" << std::endl;}
        void print(){std::cout << "A: " << c << std::endl;}
protected:
        char c;
};


class B : public A
{
public:
        B(void){c = 'b';}
        ~B(void){std::cout << "destructor B" << std::endl;}
        void print(){std::cout << "B: " << c << std::endl;}
};
```

```
int main()
{
        A* a = new A();
        B* b = new B();
        A* ptrb = b;
        a->print();
        ptrb->print();
        delete a;
        delete ptrb;
        return 0;
}
```

```
A: a
A: b
destructor A
destructor A
```

Vererbung: Polymorphismus

```
#include <iostream>
class A
{
public:
        A(void){c = 'a';}
        virtual ~A(void){std::cout << "destructor A" << std::endl;}
        virtual void print(){std::cout << "A: " << c << std::endl;}
protected:
        char c;
};


class B : public A
{
public:
        B(void){c = 'b';}
        ~B(void){std::cout << "destructor B" << std::endl;}
        void print(){std::cout << "B: " << c << std::endl;}
};
```

```
int main()
{
        A* a = new A();
        B* b = new B();
        A* ptrb = b;
        a->print();
        ptrb->print();
        delete a;
        delete ptrb;
        return 0;
}
```

```
A: a
B: b
destructor A
destructor B
destructor A
```

Vererbung: Abstract Class & Interfaces

- Abstract Class
  - Besitzt mind. eine virtuelle Methode, welche nicht implementiert wird
  - Nicht implementierte Funktion wird gekennzeichnet durch ‘= 0’
- Interface
  - Besitzt keine Attribute (lediglich Methoden)
  - Alle Methoden sind virtual und abstract
- Für mehr Informationen, siehe:

  `http://accu.org/index.php/journals/233`

Vererbung: Abstract Class & Interfaces

```
class Primitive
{
public:
        Primitive(void);
        virtual ~Primitive(void);
        virtual float intersect(Ray strahl, Vector3f &normale) = 0;
        Vector3f getColor(){return color;}
protected:
        Vector3f color;
};

class Sphere : public Primitive
{
public:
        Sphere(Vector3f c, float r, Vector3f color);
        ~Sphere(void);
        float intersect(Ray strahl, Vector3f &normale);
private:
        Vector3f c;
        float r;
};
```

Vererbung: Abstract Class & Interfaces

```
class Primitive
{
public:
        Primitive(void);
        virtual ~Primitive(void);
        virtual float intersect(Ray strahl, Vector3f &normale) = 0;
        Vector3f getColor(){return color;}
protected:
        Vector3f color;
};

class Plane : public Primitive
{
public:
        Plane(Vector3f plane_n, float s, Vector3f color);
        ~Plane(void);
        float intersect(Ray strahl, Vector3f &normale);
private:
        Vector3f plane_n;
        float s;
};
```

Vererbung: Abstract Class & Interfaces

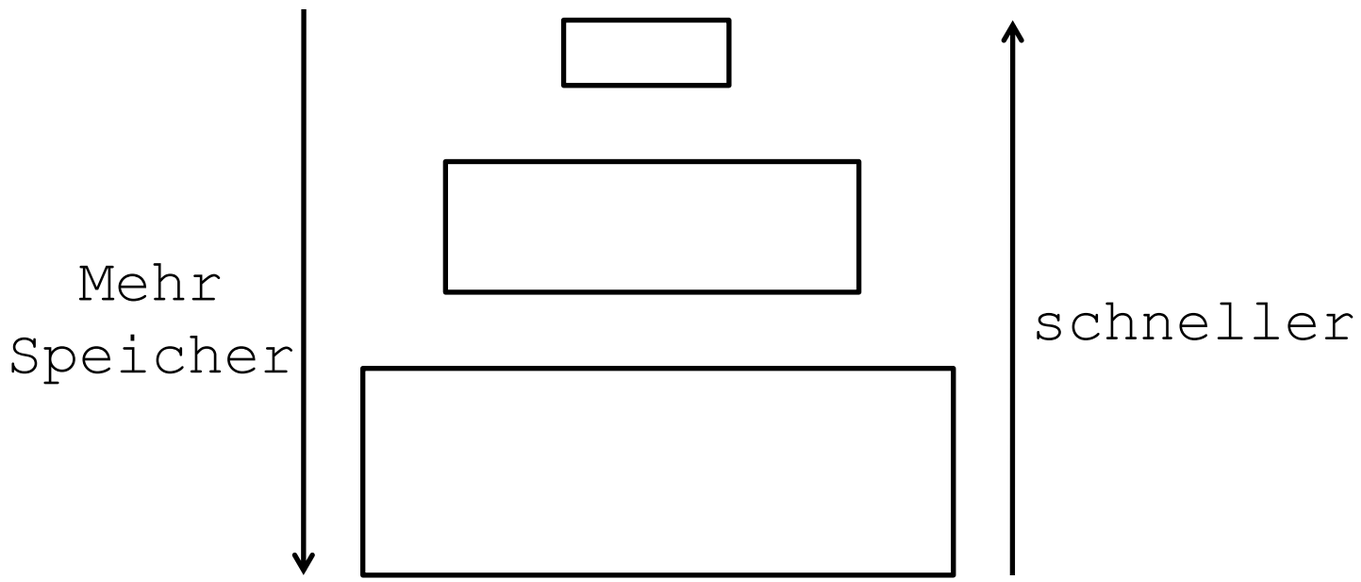
```
std::vector<Primitive*> objects;
objects.push_back(new Sphere(Vector3f(0.0f,0.0f,0.7f), 1.2f, rot));
objects.push_back(new Sphere(Vector3f(0.0f,-0.5f,2.0f), 0.8f, gruen));
objects.push_back(new Sphere(Vector3f(-1.0f,-0.5f,0.0f), 0.7f, blau));
objects.push_back(new Plane(Vector3f(0.0f,0.0f,1.0f), 0.0f, gelb));
objects.push_back(new Plane(Vector3f(0.0f,1.0f,0.0f), 3.0f, cyan));

...
```

```
Vector3f normale;
Ray strahl = generate_ray(x,y,camera);
color_rgb farbe = schwarz;
t_closest = std::numeric_limits<float>::max();
for (unsigned int i=0;i<objects.size();i++)
{
        t = objects[i]->intersect(strahl,normale);
        if (t > 0.0 && t < t_closest)
        {
                t_closest = t;
                farbe = objects[i]->getColor()*(-1*strahl.d.dot(normale));
        }
}
```

Vorbesprechung Serie 11, Aufgabe 1

- a) bis c) sind alte Prüfungsfragen
- Typo: e) Wie erklären Sie sich die Antwort in **d)**
- Cache
- Speicherhierarchie



Vorbesprechung Serie 11, Aufgabe 1

- Matrix im Speicher
  - Row-major
  - Column-major
- Matrix auf dem Stack
  - Kontinuierlich
- Matrix auf dem Heap
  - Zeilen/Spalten sind nicht zwangsläufig kontinuierlich

```
for (int row=0, tmp; row<N; ++row)
        for (int col=0; col<N; ++col)
                tmp = M[row][col];

for (int col=0, tmp; col<N; ++col)
        for (int row=0; row<N; ++row)
                tmp = M[row][col];
```

```
const int N = 500;
int M1[N][N];
```

```
int** M2 = new int*[N];
for (int i=0;i<N;i++)
{
        M2[i] = new int[N];
}
```

Vorbesprechung Serie 11, Aufgabe 2

- Übersetzen in C++ Ausdrücke
- Ohne If-Statements
- Beispiele
  - Der Integer i ist gerade
  - Der Double d ist eine ganze Zahl
  - Wenn a grösser als b ist, dann ist b grösser als c

```
int i;
(i % 2) == 0;
```

```
double d;
int(d) == d;
```

```
!(a > b && !(b > c));
!(a > b && b <= c);
  a <= b || b > c;
```

Vorbesprechung Serie 11, Aufgabe 3

- Raytracer erweitern
- Gerüst auf der Homepage
- a) neue Structs

```
struct material {
  color_rgb color;
  float reflection;
};

struct scene {
  plane *ebene;
  int nr_ebenen;
  sphere *kugel;
  int nr_kugeln;
  color_rgb hintergrund;
};
```

```
struct plane{
  Vector3f a_dir;
  Vector3f b_dir;
  Vector3f p;
  material m1;
  material m2;
};

struct sphere {
  Vector3f c;
  double r;
  material m;
};
```

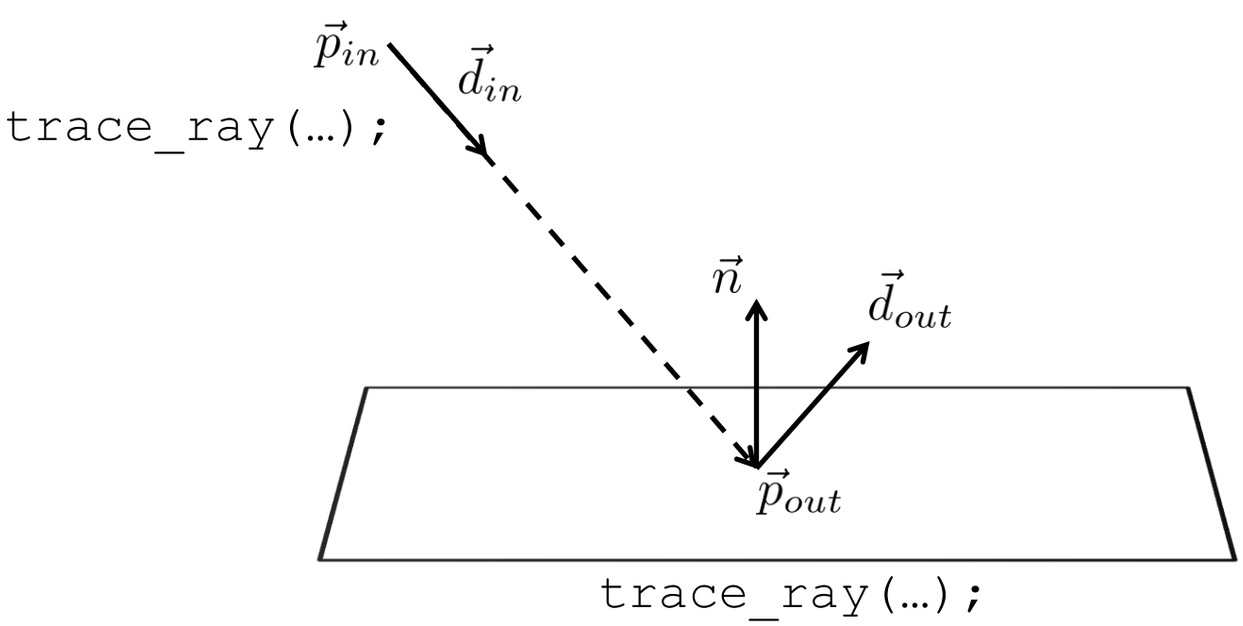
Vorbesprechung Serie 11, Aufgabe 3

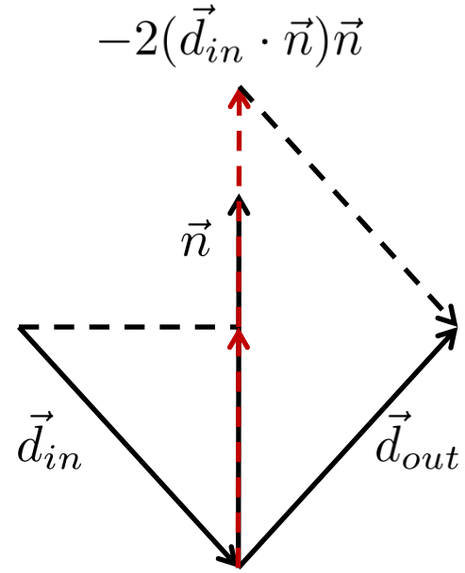
- b) trace_ray ohne Rekursion

```
color_rgb trace_ray(ray strahl, scene szene);
```

- c) trace_ray mit Rekursion

```
color_rgb trace_ray(ray strahl, int depth, scene szene);
```

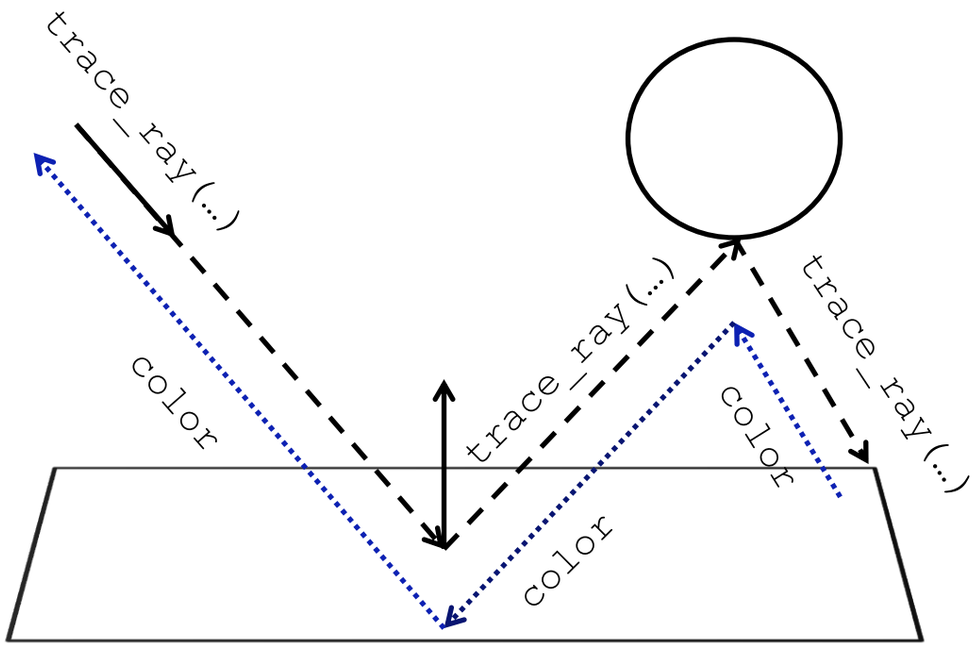


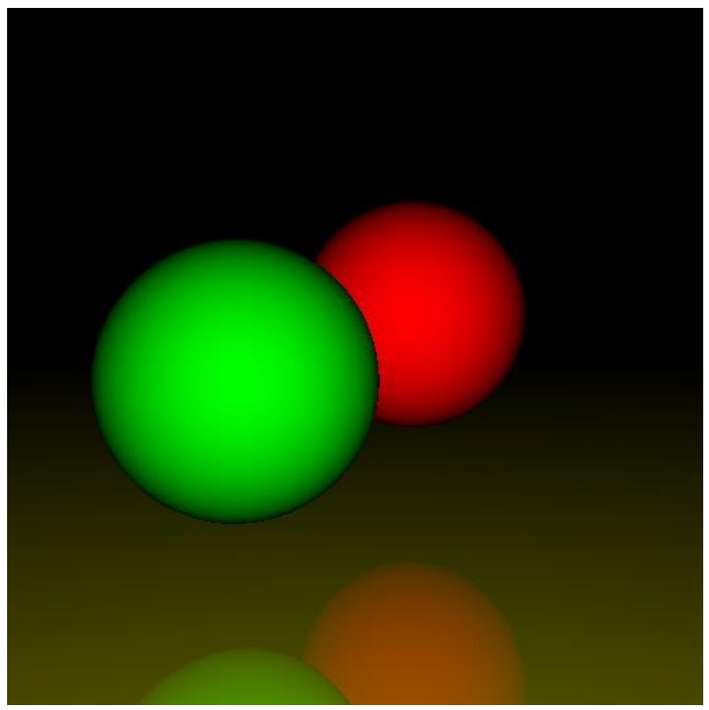


Vorbesprechung Serie 11, Aufgabe 3

- d) Koeffizient für Reflexion
  - r zwischen 0 und 1
  - Reflexionskoeffizient == 0: Rekursion abbrechen

$$f = f_{object} * (1 - r_{objekt}) + f_{spiegelung} * r_{objekt}$$



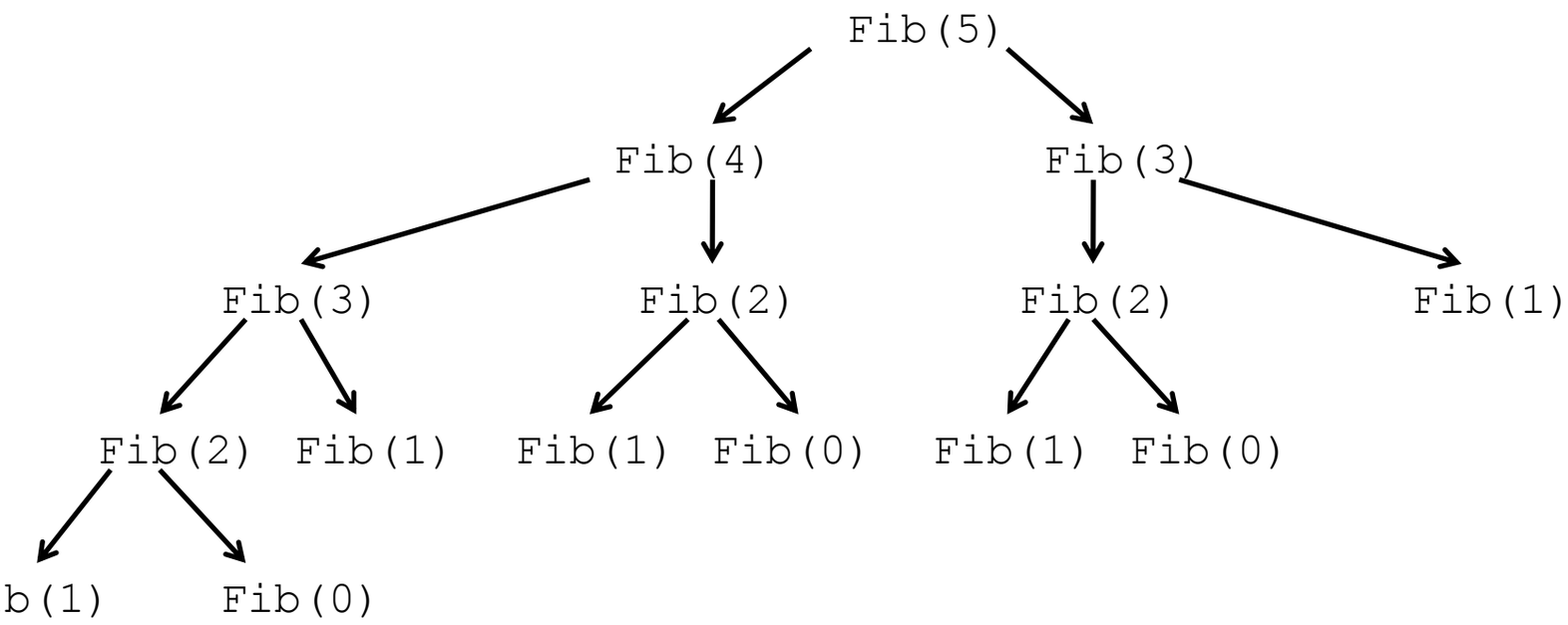


Vorbesprechung Serie 11, Aufgabe 3

- e) Neue Körper, Positionen ändern
- f) Neue Lichtquellen und Eigenschaften
  - Schatten benötigen Lichtquellen
  - Transparenz: siehe Snell‘s Law
  - Halbschatten: Lichtquelle mit Ausdehnung
  - …

http://www.povray.org/

Nachbesprechung Serie 9



- Konservative Schätzung der Anzahl Funktionsaufrufe für fib(n):

$$t(n) = 1 + t(n-1) + t(n-2) > 2t(n-2)$$

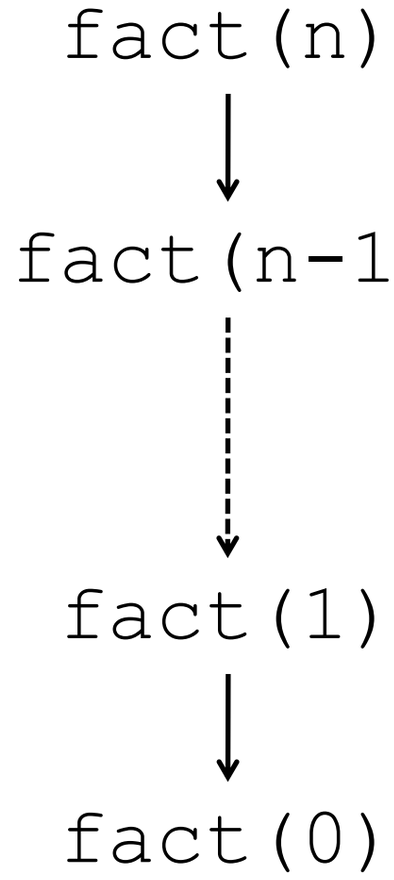
$$t(n) > 2t(n-2) > 2^{\frac{n}{2}} t(0) = \sqrt{2}^{n}$$

Nachbesprechung Serie 9

- Rekursion ist nicht per se langsam
- Beispiel Fakultät:

$$fact(0) = 1$$

$$fact(n) = n fact(n-1)$$



- Anzahl Funktionsaufrufe:

$$t(n) = 1 + t(n-1) = n + t(0) = n + 1$$

Nachbesprechung Serie 9

- Stack overflow

```
int sum(int n)
{
        if (n==0) return 0;
        else return n+sum(n-1);
}
```

```
int main()
{
        std::cout << sum(5000) << std::endl;
        return 0;
}
```

- Inline nicht möglich