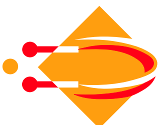
An Introduction to Graphics and More on Panda3d, C++, and Python

Kim Steenstrup Pedersen

kimstp@itu.dk

www.itu.dk/courses/SSPG/F2005/

The IT University of Copenhagen



Computer Graphics

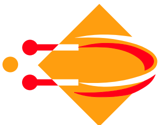
Model frame

World frame

Screen frame

Camera frame

Computer graphics: Defining a model, positioning and orienting the camera, positioning the light sources.

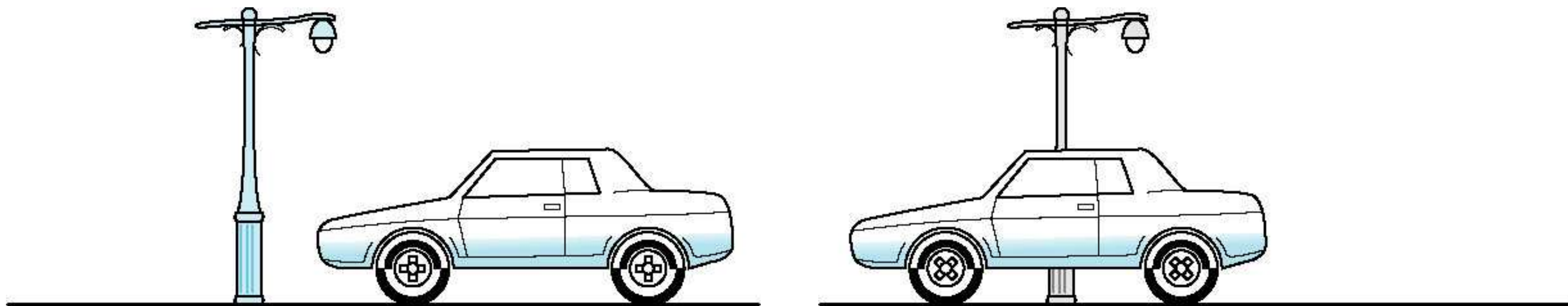


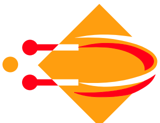
Hierarchical models

Modelling complex objects through simple objects often leads to hierarchical relationships between the sub-objects, hence a hierarchical model.

We can represent hierarchical models by data structures such as trees and directed acyclic graphs (DAG).

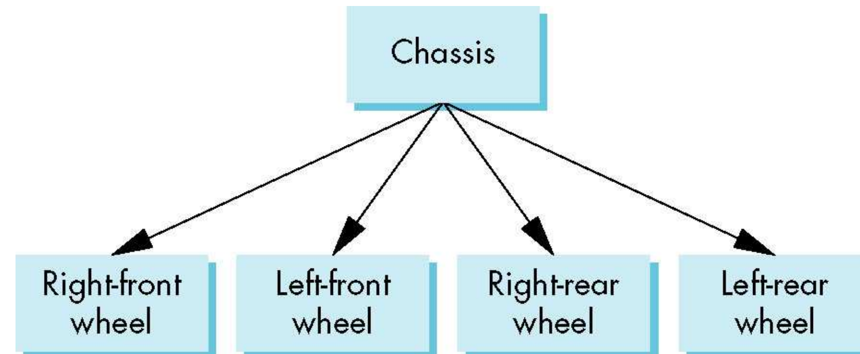
Example: Animation of a car model



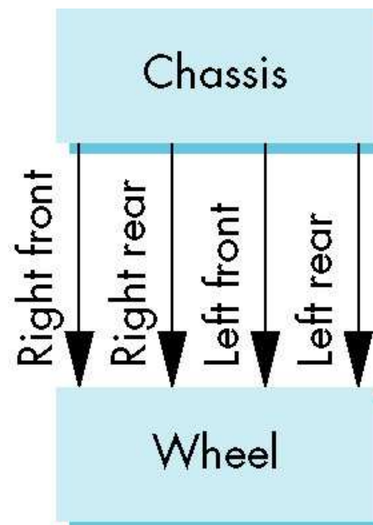


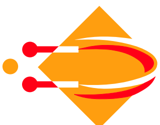
Hierarchical representation of the car

Representing the car in a tree structure:



Representing the car in a DAG structure (only one instance of the wheel):



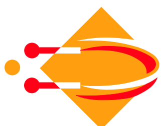


Scene graphs

A scene consisting of objects, light sources, and cameras each with properties such as position, colour, etc. can also be represented as a graph or tree.

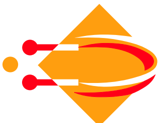
A scene graph example:

- Scene
  - Separator
    - Color
    - Translate
      - Rotate
    - Object 1
      - Translate
        - Object 2
  - Separator
    - Translate
      - Rotate
    - Object 3



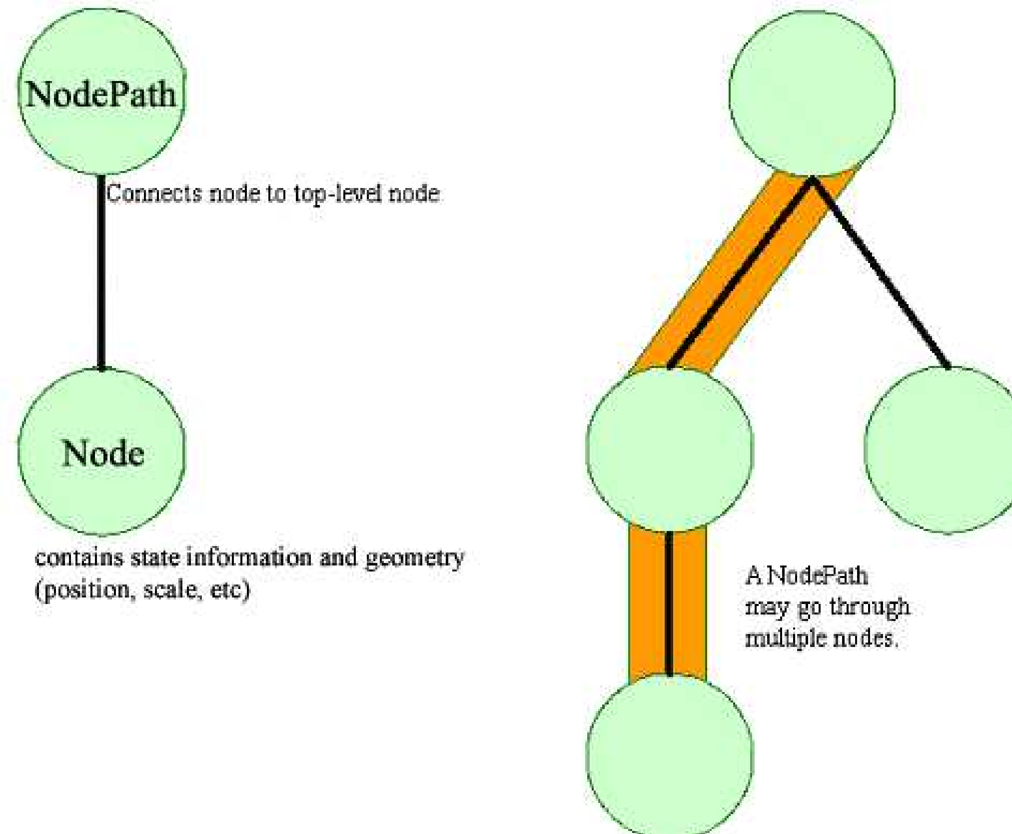
Scene Graphs in General: Summary

- Complex objects and scenes have a hierarchical structure and are therefore well described using data structures such as trees and DAG's.
- The benefit of a data structure description like scene graphs is that we can dynamically update our scene by adding, editing, or deleting subparts of the scene graph.
- The way we should traverse a scene graph is tightly connected with how it is build.

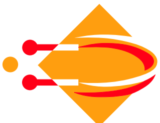


Scene Graphs in Panda3d: Nodes and NodePath

Basic units: Node and NodePath



- A Node stores the geometry and properties of models (position, orientation, colour, texture etc.)
- A NodePath connects two or more Nodes and forms the hierarchy of the scene.
- A NodePath is the interface to manipulate Nodes and the scene graph.



Scene Graphs in Panda3d: Special Nodes

Important predefined top-level nodes:

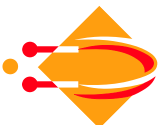
render: Default 3D scene graph. Will be rendered (drawn).

render2d: Default 2D scene graph. Will be rendered.

aspect2d: Child of render2d that will scale the underlying graph to compensate for the non-square aspect ratio of the screen.

hidden: An ordinary Node, not parented to render and will not be rendered.

Only nodes connected to render or render2d will be rendered.



Scene Graphs in Panda3d: Creating New Nodes

Loading a model:

```
<NodePath> = loader.loadModel('<filename>')
```

Creating empty nodes:

```
<NodePath> = <NodePath>.attachNewNode('<nodeName>')
```

Create new NodePath and Node to subpart of model:

```
<NodePath> = <ParentNodePath>.find('<NameOfPart>')
```

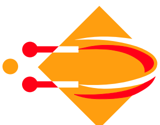
Setting the parent:

```
<NodePath>.reparentTo(<NodePath>)
```

Show example!

Note:

> `print` <NodePath> in python prints the names of the nodes that the NodePath spans.



Scene Graphs in Panda3d: Searching for Nodes

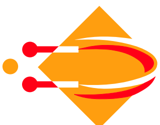
Search for a Node by name or type in the graph:

```
<NodePath> = <NodePath>.find("<path>")
<NodePathCollection> = <NodePath>.findAllMatches("<path>")
```

`<path>` is a pattern matching string.

Example:

```
# Find a node named 'MyDummy'
environ.find('MyDummy')
# Searching for nodes of type GeomNode
coll=environ.findAllMatches('/+GeomNode')
print coll
print coll[0]
```



Scene Graphs in Panda3d: Common State Changes

Set position, orientation and scale of all nodes in NodePath:

```
<NodePath>.setPos(<x>,<y>,<z>)
<NodePath>.setHpr(<h>,<p>,<r>)
<NodePath>.setScale(<sx>,<sy>,<sz>)
<NodePath>.lookAt(<NodePath>)
```

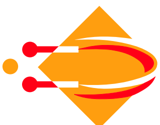
Get position, orientation and scale of nodes in NodePath:

```
<NodePath>.getPos()
<NodePath>.getHpr()
<NodePath>.getScale()
```

Use `<NodePath>.hide()` to hide a Node and all its children. Will not be rendered. To make visible again use `<NodePath>.show()`.

Note:

> The sceneEditor and Scene graph browser can be useful tools for building, manipulating, and viewing scene graphs. Need to modify `Config.prc`.



More C++: Constructors and other class details

```
class ModelClass {
 public:
  ModelClass() {}; // Default constructor
  ModelClass(std::istream& in);
  std::vector<Vertex> getVertices() const;
  ...
 private:
  int modelSize;
  std::vector<Vertex> vertices;
};
```

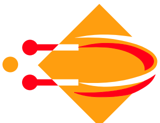
Default initialisation (synthesised constructors)

Constructor initialisers

```
    ModelClass::ModelClass() : modelSize(0) {}
```

Things to note:

- `const` methods.
- Constructors can have several arguments, some constructors are special like the default constructor (more about this later).



Struct

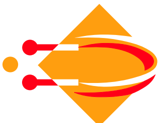
C++ supports a type called `struct` for backwards compatibility with C. In C++ a `struct` acts as a class except for default protection labels.

This code is equivalent:

```
class ModelClass {
 public:
  ModelClass() {};
  void loadModel();
 private:
  std::vector<Vertex> vertices;
};

struct ModelClass {
 public:
  ModelClass() {};
  void loadModel();
 private:
  std::vector<Vertex> vertices;
};
```

Things to note:

- The default protection label for a class is `private` and for a struct it is `public`.



STL = Standard Template Library

STL provides you with a lot of useful types, container classes, and algorithms. All this is in the namespace called `std`.

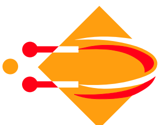
Examples of types and containers:

```
#include <iostream> ...  std::cout
#include <string>   ...  std::string
#include <vector>   ...  std::vector
#include <list>     ...  std::list
```

Examples of algorithms working on containers:

```
#include <algorithms>
...
sort(...);
find(...);
```

Useful, but watch out for varying performance between implementations of STL.

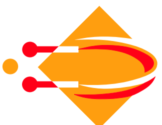


Sequential and random access containers

Problem: Dissect student list according to pass and fail.

```
// predicate to determine whether a student failed
bool fgrade(const Student_info& s)
{
        return grade(s) < 60;
}
...
vector<Student_info> extract_fails
                     (vector<Student_info>& students)
{
        vector<Student_info> pass, fail;

        for (vector<Student_info>::size_type i = 0;
             i != students.size(); ++i)
                if (fgrade(students[i]))
                        fail.push_back(students[i]);
                else
                        pass.push_back(students[i]);
        students = pass;
        return fail;
}
```

Things to note:



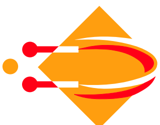
2 copies of data!

Removing elements from vectors

```
// second try: correct but potentially slow
vector<Student_info> extract_fails
                     (vector<Student_info>& students)
{
   vector<Student_info> fail;
   vector<Student_info>::size_type i = 0;

   // invariant: elements ‘[0,’ ‘i)’ of ‘students’ represent
   // passing grades
   while (i != students.size()) {
      if (fgrade(students[i])) {
         fail.push_back(students[i]);
         students.erase(students.begin() + i);
      } else
         ++i;
   }
   return fail;
}
```

Things to note:

Quadratic computational complexity for deletion, sequential versus random access.

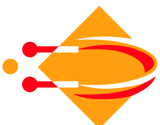


Iterators

```
// version 3: iterators but no indexing; still potentially slow
vector<Student_info> extract_fails
                     (vector<Student_info>& students)
{
   vector<Student_info> fail;
   vector<Student_info>::iterator iter = students.begin();

   while (iter != students.end()) {
      if (fgrade(*iter)) {
         fail.push_back(*iter);
         iter = students.erase(iter);
      } else
         ++iter;
   }
   return fail;
}
```

Things to note:

Iterators, dereference operator `*`, `(*iter).name` ⇔ `iter->name` ,
invalid references after `erase`

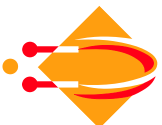


std::list

```
// version 4: use 'list' instead of 'vector'
list<Student_info> extract_fails(list<Student_info>& students)
{
   list<Student_info> fail;
   list<Student_info>::iterator iter = students.begin();

   while (iter != students.end()) {
      if (fgrade(*iter)) {
         fail.push_back(*iter);
         iter = students.erase(iter);
      } else
         ++iter;
   }
   return fail;
}
```

Things to note:

- Linear versus quadratic computational complexity for deletion of elements!

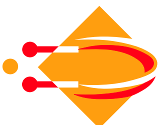


Iterators and standard library algorithms

```
bool space(char c) { return isspace(c); }

bool not_space(char c) { return !isspace(c); }

vector<string> split(const string& str)
{
        typedef string::const_iterator iter;
        vector<string> ret;

        iter i = str.begin();
        while (i != str.end()) {
                i = find_if(i, str.end(), not_space);
                iter j = find_if(i, str.end(), space);
                if (i != str.end())
                        ret.push_back(string(i, j));
                i = j;
        }
        return ret;
}
```

Things to note:

Iterators, `find_if`, problem with overloading of `isspace()` (versions for `char` and `wchar_t`)



Static and find

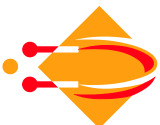
```
bool not_url_char(char c)
{
  // characters, in addition to alphanumerics, that can
  // appear in a URL
  static const string url_ch = "~;/?:@=&$$-_.+!*'(),";

  // see whether 'c' can appear in a URL and return the negative
  return !(isalnum(c) ||
       find(url_ch.begin(), url_ch.end(), c) != url_ch.end());
}
```

Other example:

```
int numberOfCalls() {
  static int count = 0;
  return ++count;
}
int main() {
  for (int i = 0; i < 10; i++)
    cout << numberOfCalls() << endl;
}
```

Things to note:

`static`, `find`, other algorithms exists in `<algorithm>` and `<numeric>`



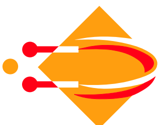
Data structure independence by iterators

```
find(c.begin(), c.end, val)
...
c.find(val)
...
find(c,val)
```

- Template `find` works on any contiguous part of any container.
- `c.find()` would require `find` to be implemented as part of `c`.
- Would not work on `build-in` type `array`.
- `find(c,val)` would not delimit range

Advantages of half-open intervals: `v.begin(),v.end()`.

- `v.end()` defined for empty containers
- Only equality and inequality operations required
- `v.end()` is a natural out-of-range value.



Memory, Variables, and Pointers

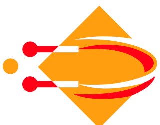
No shielding from dealing with computer memory in C++!
Memory can be thought of as a long list of “slots”

Slot 0
Slot 1
Slot 2
⋮

Each slot holds 1 byte $\simeq$ 8 bits.
Some common variables and typical sizes:

`char`	`short`	`int`	`long`	`float`	`double`
8bit	16bit	16bit	32bit	32bit	64bit

Machine dependent!



Pointers

Pointers are also variables that point to other variables.

What:	Value	$\longrightarrow$	Variable
Where:	Address	$\longrightarrow$	Pointer

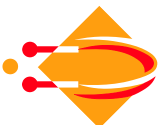
Foo.C

```
int main() {
  int A = 10000;
  int *p = &A;
  cout << A << endl;  // 10000
  cout << p << endl;  // Address: bffff600
  cout << *p << endl; // 10000
  cout << &A << endl; // Address: bffff600
  cout << &p << endl; // Address: bffff604
}
```

Things to note:

Address operator `&`, dereference operator `*`

Warnings: Pointers may be changed.



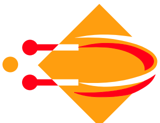
Pointers to objects

Foo.H

```
class Foo {
public:
  Foo();
  Foo(int a, int b);
  ~Foo();
  void bar();
  int blah;
};

Foo myFooInstance = new Foo(0,0); // Wrong!

Foo *myFooInstance = new Foo(0,0);
```

To call methods through a pointer use:

```
myFooInstance->bar();   myFooInstance->blah = 5;
```



References

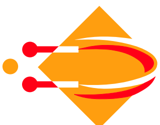
main.C

```
int main() {
  int foo = 10;
  int &bar = foo;

  bar += 10;
  cout << "foo is: " << foo << endl;
  cout << "bar is: " << bar << endl;

  foo = 5;
  cout << "foo is: " << foo << endl;
  cout << "bar is: " << bar << endl;
}
```

Output:

```
foo is: 20
bar is: 20
foo is: 5
bar is: 5
```

Warnings: Only assignable at creation



Memory management

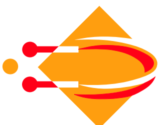
Declaring and using variables is a major aspect of programming.

Variable type	Local	Global
Validity	Scope {. . . }	Program
Typical stored	the Stack	the Heap

scope.C

```
{
  int myInteger; // allocated memory for myInteger
  // myInteger is available
  {
    Bar bar; // allocated memory for bar
    // bar and myInteger is available
  } // deallocated bar
  // only myInteger is available
} // deallocated myInteger
```

Warning: Don't use local variables outside their scope.

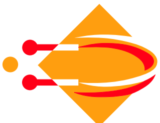


Memory management:Memory leaks

global.C

```
{
  Bar *p;

  p = new Bar();
  // What p is pointing to is valid
  // outside the current scope
  ...
  // p must be deallocated manually!
  delete p;
}
```

Warnings: Don’t `delete` what you didn’t `new`; Avoid memory leaks.



Arrays

Arrays in C++ are cumbersome.

Syntax

Dynamically allocated:

```
<type> *arrayName = new <type>[number elements]
```

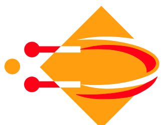
or statically allocated

```
<type> arrayName[number elements]
```

```
{
  int* integerArray = new int[100];
  integerArray[3] = 255;

  int integerArray2[100];
  integerArray2[3] = 255;
}
```

Elements are stored consecutively, i.e.

`integerArray[3] = 255;` is equivalent to
`*(integerArray + 3) = 255;`



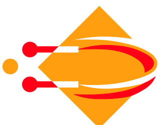
Warning Be careful only to index what you've allocated!

Copy control

```
class Vec {
public:
   Vec() { create(); }                  // Default constructor
   Vec(const Vec& v);                   // Copy constructor
   Vec& operator=(const Vec& rhs); // Assigment operator
   Vec(size_type n);                    // Ordinary constructor
   ...
private:
   double* data;
}
...
Vec v;       // initialization (default constructor)
Vec v2 = v; // initialization (copy constructor)
Vec v3(10); // initialization (constructor)
v2 = v3;     // assignment (operator=)
```

Things to note:

copy constructor initialises object, assignment operator = obliterates data in lhs object and copies rhs object into lhs

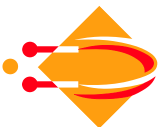


Copy control: Assignment implementation

```
Vec& Vec::operator=(const Vec& rhs) {
  // check for self-assigment
  if (&rhs != this) {
    // copy data in rhs into this Vec.
  }
  return *this;
}
```

Things to note:

- Pointer to current object `this`



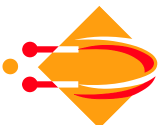
Defining your own operators

Operators in general: <type> `operator`OP(<arguments>)

```
double operator+(const double& a, const double& b) { ... }
double operator*(const double& a, const double& b) { ... }
```

Things to note:

Operators can be either member or nonmember functions of classes depending on whether they change their left operand or not.



Destructor's and the rule of three

```
class Vec {
public:
    ~Vec() {  // Destructor
      // destroy what ever data dynamically allocated by Vec
    }
    ...
}
```

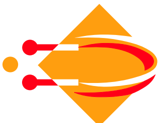
Things to note:

The destructor is called when objects are deleted or go out of scope, and should deallocate dynamically allocated memory.

The rule of three

Classes that manage resources such as memory require close attention to copy control. You need to define the following for such a class T:

```
T::T()          // One or more constructors, perhaps with
                // arguments
T::~T()         // The destructor
T::T(const T&) // The copy constructor
T& T::operator=(const T&) // The assignment operator
```



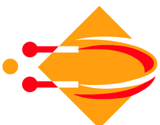
Inheritance: The base class

Base class in geometry.h:

```
#ifndef __GEOMETRY__
#define __GEOMETRY__
#include <vector>

class Geometry {
 public:
  void draw() {};

 protected:
  std::vector<std::vector<double> > vertices;
};
#endif
```

Things to note:

The use of `#ifndef` and `#define` in order to avoid repeated definitions of types, `protected` access



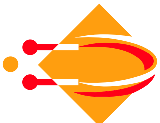
Inheritance: Derived classes

point.h:

```
#ifndef __POINT__
#define __POINT__
#include "geometry.h"

class Point : public Geometry {
 public:
  Point(double x, double y);
  void draw();
};
#endif
```

Things to note:

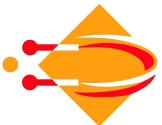
- `public` derived class, inheritance of `public` and `protected` methods and variables from base class (`Geometry`), overwriting inherited methods (`draw`), multiple inheritance is possible



Inheritance: Derived classes

point.cpp:

```
#include "point.h"
using std::vector;

Point::Point(double x, double y) {
  vector<double> v;
  v.push_back(x);
  v.push_back(y);
  vertices.push_back(v);
}

void Point::draw() {
 // Code for drawing a point should be added here
}
```

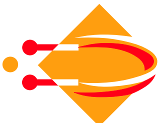


Polymorphism and virtual functions

```
class Geometry {
 public:
  virtual void draw() {};

 protected:
  std::vector<std::vector<double> > vertices;
};
```

Things to note:

`virtual` method definition

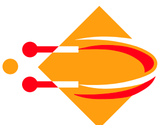


Going through a list of Geometry objects

```
int main() {
  vector<Geometry*> geomlist;

  Point *p = new Point(1,2);
  geomlist.push_back(p);

  Line *l = new Line(1,2,4,5);
  geomlist.push_back(l);

  Triangle *t = new Triangle(1,2,3,4,5,6);
  geomlist.push_back(t);

  vector<Geometry*>::iterator iter = geomlist.begin();
  while (iter != geomlist.end()) {
    (*iter)->draw();
    iter++;
  }
  return 0;
}
```

Things to note:

- `vector` of pointers to `Geometry` objects, `draw` method resolved at run-time

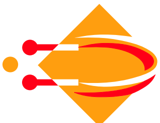


Abstract classes

```
class Geometry {
 public:
  virtual void draw() = 0;

 protected:
  std::vector<std::vector<double> > vertices;
};
```

Things to note:

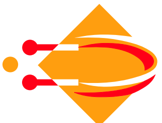
- Pure virtual draw method, abstract classes can't be instantiated



More Python: References to functions

References to functions:

```
def foo():
    print "Foo!"

f = foo
foo()
f()

def foo():
   i = 0
   while i < 10:
      print "Foo!"
      i = i + 1

foo()
f()
del f
f() # Error!
```

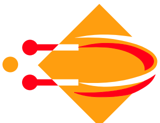


More Python: Modules

- A module is a file containing Python definitions and statements.
- Module name given by the file name.

Example:

```
In foo.py:
def bar():
   print "foobar"

def goo():
   print "goo"
...
import foo
foo.bar()
...
from foo import bar
bar()
foo.goo()
...
from foo import *
bar()
goo()
```

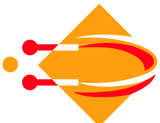


More Python: Packages

- A package is a collection of Python modules.
- Package names given by directory structure.
- All package directories must include a file called `__init__.py` for initialisation.

Example:

```
nodes/
    __init__.py
    geometry/
        __init__.py
        triangle.py
    transformation/
        __init__.py
        rotate.py
...
import nodes.geometry.triangle
```



More Python: Container types

Lists:

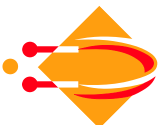
```
a=[1, 'Hello', 4.5]
a.append(42)
del a[0]
del a
```

Tuples:

```
t=1, 2, 'games'
t[0]
u=t,(0,1,2) # nested tuples
x,y,z=t     # tuples assignment
```

Dictionaries (associative arrays):

```
colors = {'blue': 1, 'red': 2}
colors['blue']
colors.keys()
colors.has_key('red')
colors.has_key('yellow')
```

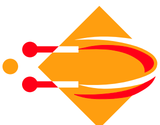


More Python: Classes

```
class MyClass:
   "A simple class"
   i=12345
   def f(self,a):
      return 'Hello world'
   def __init__(self):  # Constructor
      self.data=[] # Creates a list object

x=MyClass() # Create an object
x.f(2)
x.i
x.data

class FooClass:
   pass

class AnotherClass(MyClass,FooClass):
   a = 42
```

Things to note:

> Help string, `self` (`this` pointer), methods should take at least one argument `self`, `pass` (empty statement), inheritance (multiple inheritance allowed).



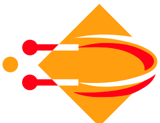
Extending Python with C++

`hellomodule.cpp`:

```
#include <Python.h>
#include <iostream>

// Forward definition of module initialization
extern "C" __declspec(dllexport) void inithellomodule();

// Forward definition of external linkable C++ functions
extern "C" static PyObject*
      hellomodule_helloworld(PyObject* self, PyObject* args);


// Python method table for hellomodule module
static PyMethodDef HellomoduleMethods[] = {
        {"helloworld", hellomodule_helloworld, METH_VARARGS,
         "Prints a greeting"},
        {NULL, NULL, 0, NULL}
};
```

Things to note:

`#include <Python.h>`, module initialisation, `extern` "C", `__declspec(dllexport)`, `static` functions and data, `PyObject*`, method table `PyMethodDef`.



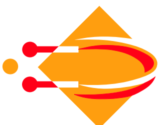
Extending Python with C++

`hellomodule.cpp`:

```
/* Implementation of methods of the hellomodule module */

// init method is the only non-static function in the module
void inithellomodule(void) {
    (void) Py_InitModule("hellomodule", HellomoduleMethods);
}

static PyObject*
        hellomodule_helloworld(PyObject* self, PyObject* args)
{
    std::cout << "Hello World" << std::endl;
    Py_INCREF(Py_None);
    return Py_None;
}
```

Things to note:

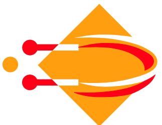
- `Py_InitModule`, `Py_None` object (empty object), `Py_INCREF`.
- Build as a dynamically linked library (dll) called hellomodule.dll.



Extending Python with C++

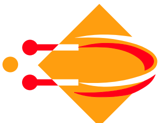
`testhellomodule.py`:

```
import hellomodule
hellomodule.helloworld()
```



Overview of today

- Basic concepts of computer graphics.
- Scene graphs in general and in Panda3d.
- C++: STL containers, pointers, memory management and inheritance and polymorphism.
- Python: Functions, modules, packages, containers, classes.
- Extending Python using C++, part I: Functions.

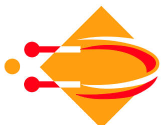


Assignment due February 25

Exercises today in rooms 0A17, 3A50, and 3A54. We will all meet in room 0A17 (multimedia lab) to get general information.

Your assignment for February 25 will be to get acquainted with C++ as well as programming Python extensions in C++:

1. Write a C++ class that implements a simple queue. That is, a first in first out (FIFO) list. For simplicity assume that the queue can hold only objects of type int. This class should at least have a method for adding a new element to the queue (to the back) and removing an element from the queue (the first element). Include whatever other methods you find necessary.
   (Hint: You can use `std::list` and maybe iterators.)
   (Advanced: If you know about template classes in C++ you could write the queue as a template class allowing your queue any type of object.)
2. Write a Python extension in C++ so that you can use your queue class inside of Python. That is, it should be possible to create an object instance of your queue class inside of Python and use the methods of the class on this object.
   (Advanced: If you implemented the template version of the queue, you might want to force the queue to only hold `PyObject*`, i.e. so that it can hold any Python object.)



Reading material

Reading material for this lecture:

- Panda doc: V.A Scene graphs
- KM: Ch. 5-6, 9-11, 13
- Python tutorial
- Python extension documentation

Reading material for the next lecture:

- JG: Ch. 3-5
- Parts of Panda documentation (to be announced)
- Python extension documentation