Extreme Object Models Using Boost.Python

BoostCon 2007

May 15, 2007

Timothy M. Shead <tshead@sandia.gov>
Sandia National Laboratories
Data Analysis & Visualization, 1424

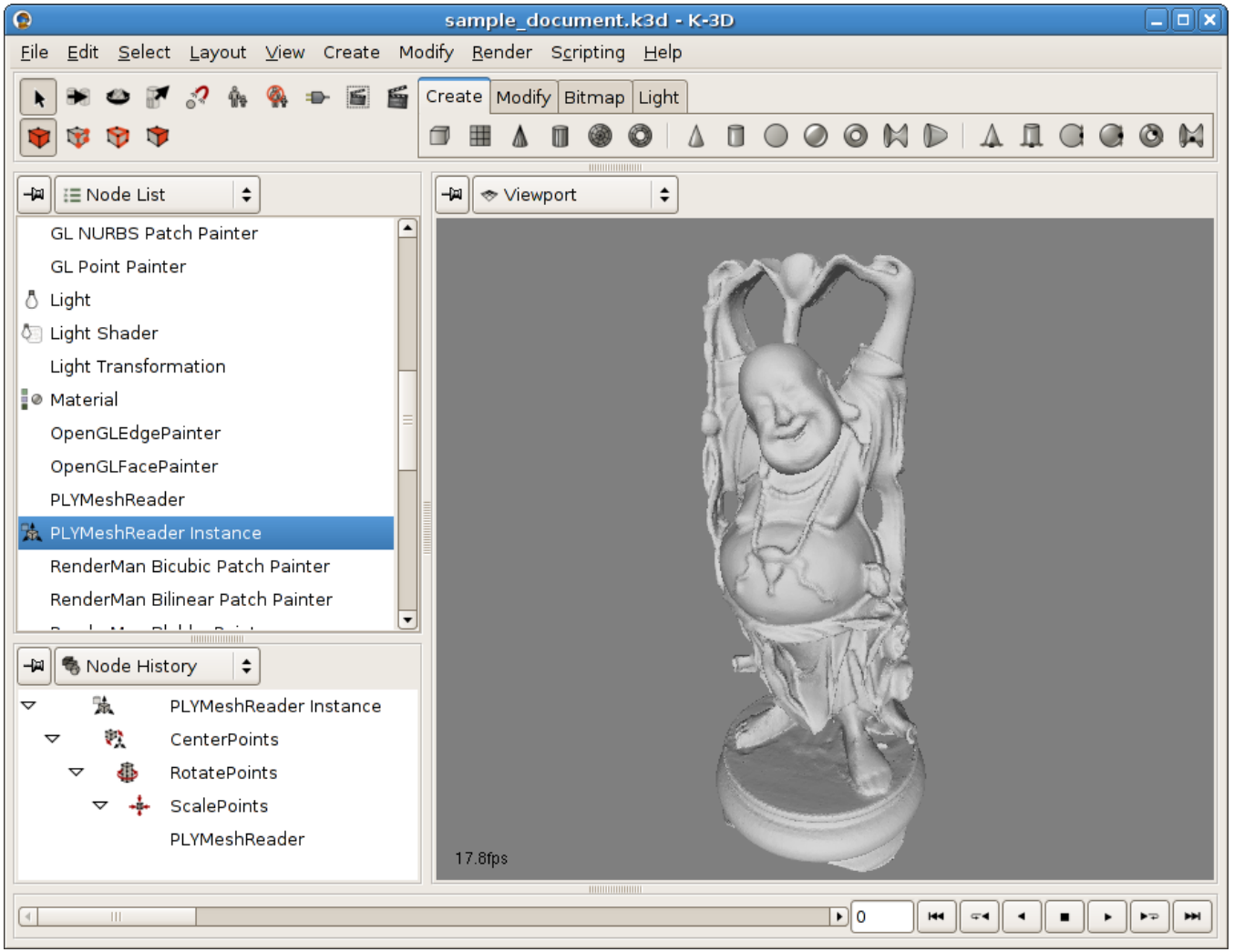
Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This document is SAND Number: 2007-2957 C

What the heck is *Extreme?*

Mainly, it's “real world messiness”:

- **I don't want my Python objects to be identical to their C++ counterparts.**
- **I need to implement my own “special” Python methods.**
- **I need to wrap C++ templates.**
- **I need to embed Python, rather than extending it.**
- **I need to provide functions that can accept / return multiple object types.**

K-3D - http://www.k-3d.org



Data: http://graphics.stanford.edu/data/3Dscanrep/ "Happy Buddha"

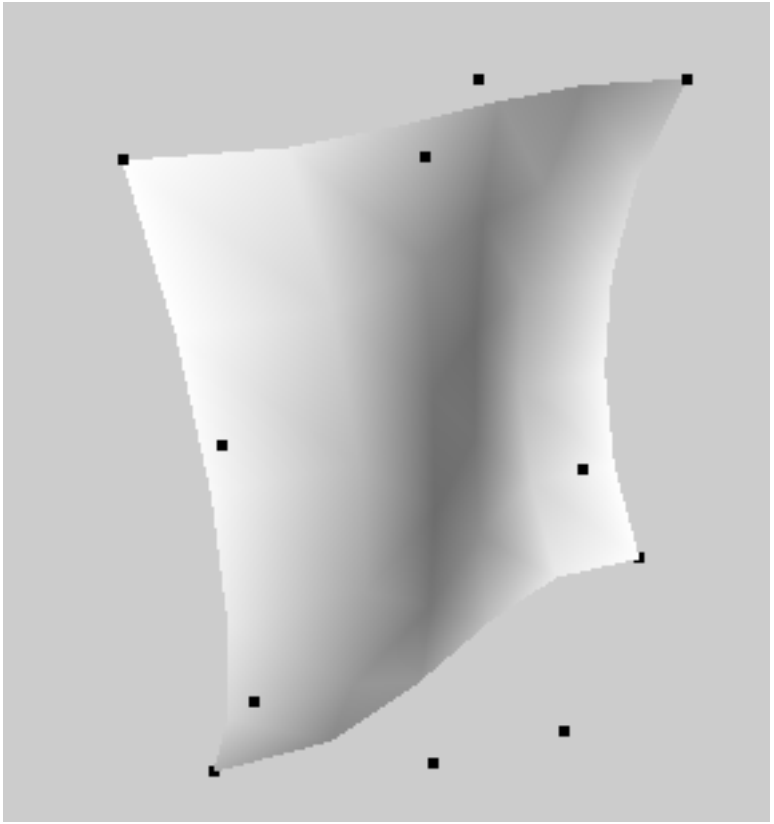
Sandia National Laboratories

Automating repetitive tasks

create_bicubic_patch.py

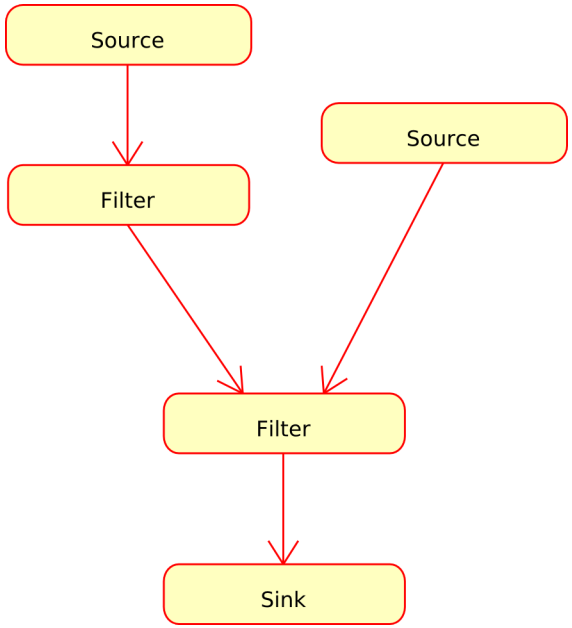
File Edit

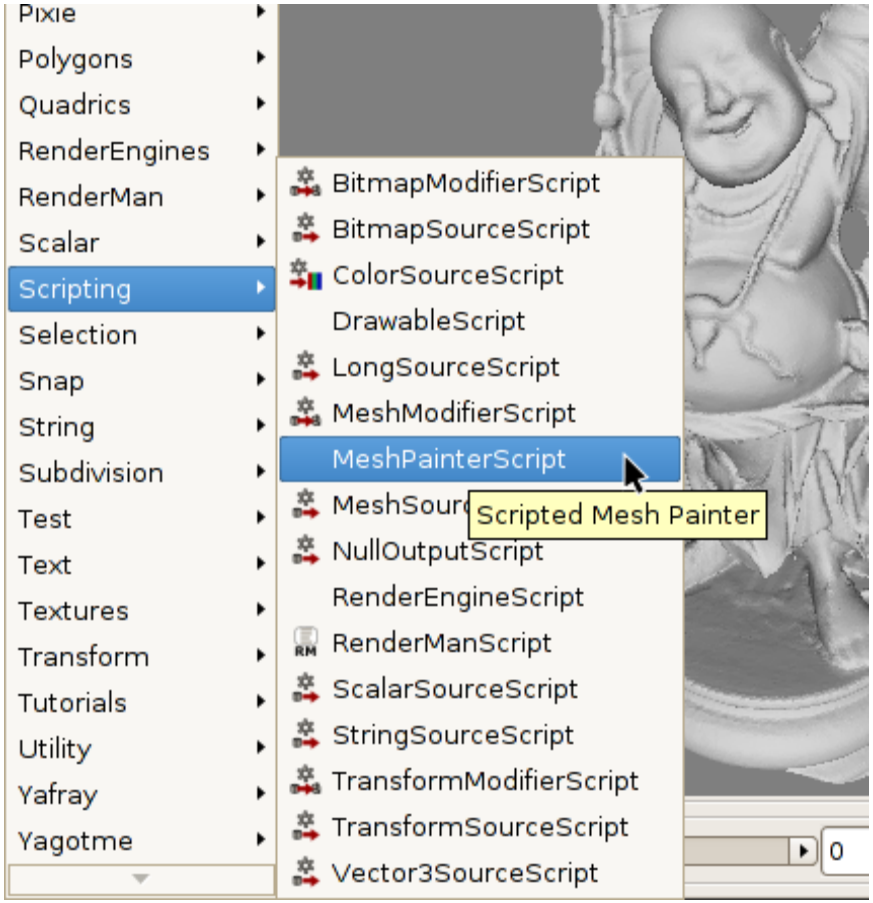
```
import k3d

doc = Document
doc.start_change_set()
try:
	material = doc.new_node("RenderManMaterial")
	material.name = "Patch Material"
	material.color = k3d.color(1, 1, 1)

	frozen_mesh = doc.new_node("FrozenMesh")
	frozen_mesh.name = "Bicubic Patch"

	mesh = frozen_mesh.dynamic_cast("imesh_storage").new_mesh()

	positions = [
		(-5, -5, 0), (-2, -5, 2), (2, -5, -2), (5, -5, 0),
		(-5, -2, 2), (-2, -2, 5), (2, -2, -5), (5, -2, -2),
		(-5, 2, 2), (-2, 2, 5), (2, 2, -5), (5, 2, -2),
		(-5, 5, 0), (-2, 5, 2), (2, 5, -2), (5, 5, 0)
		]

	points = mesh.create_points()
	point_selection = mesh.create_point_selection()

	for position in positions:
		points.append(k3d.point3(position[0], position[2], -position[1]))

	bicubic_patches = mesh.create_bicubic_patches()

	patch_selection = bicubic_patches.create_patch_selection()
	patch_selection.append(0)

	patch_materials = bicubic_patches.create_patch_materials()
	patch_materials.append(material.dynamic_cast("imaterial"))
```



Sandia National Laboratories

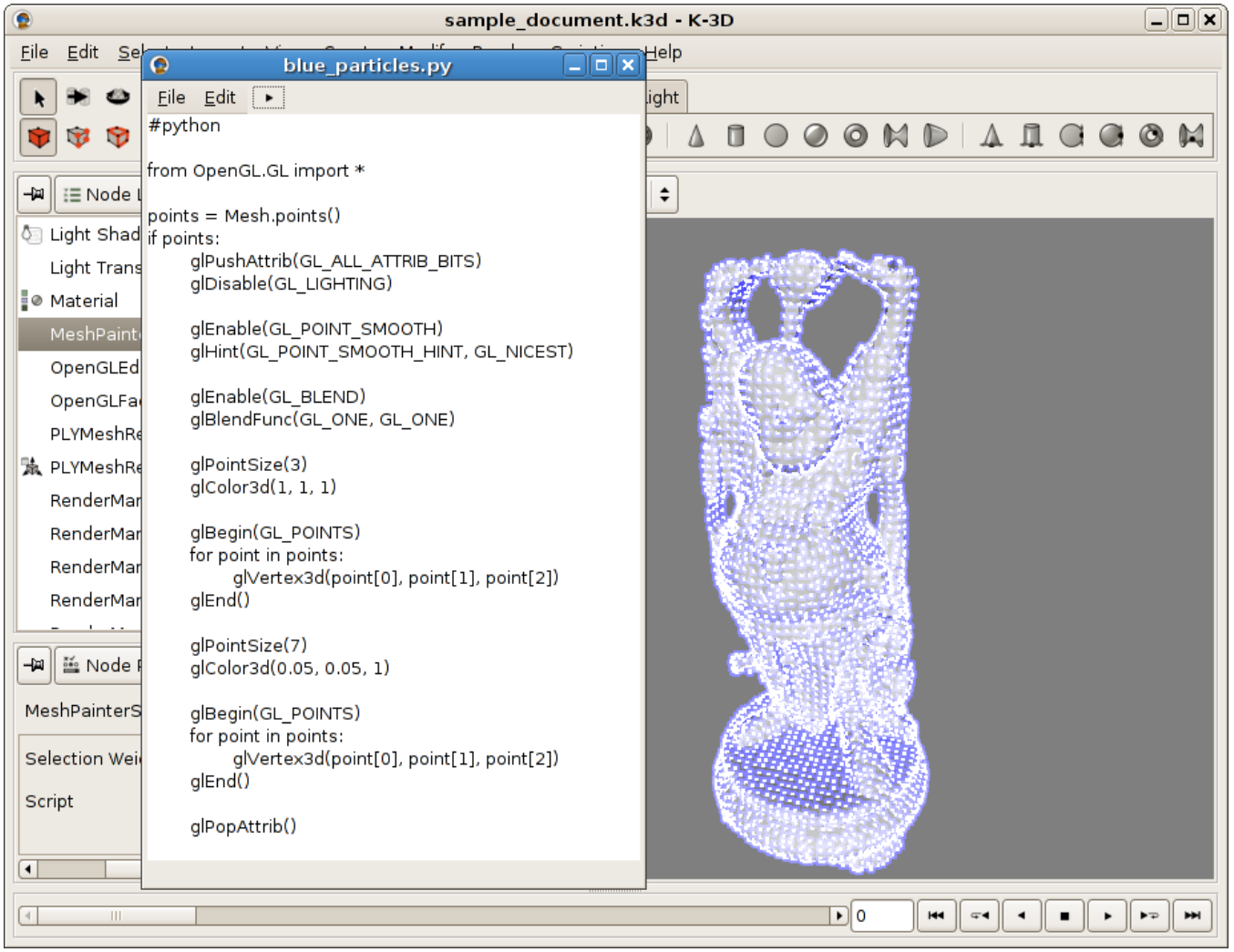
Python pipeline components





Sandia National Laboratories

Python rendering



Sandia National Laboratories

Python regression test

tshead@joe:~/k3d/tests

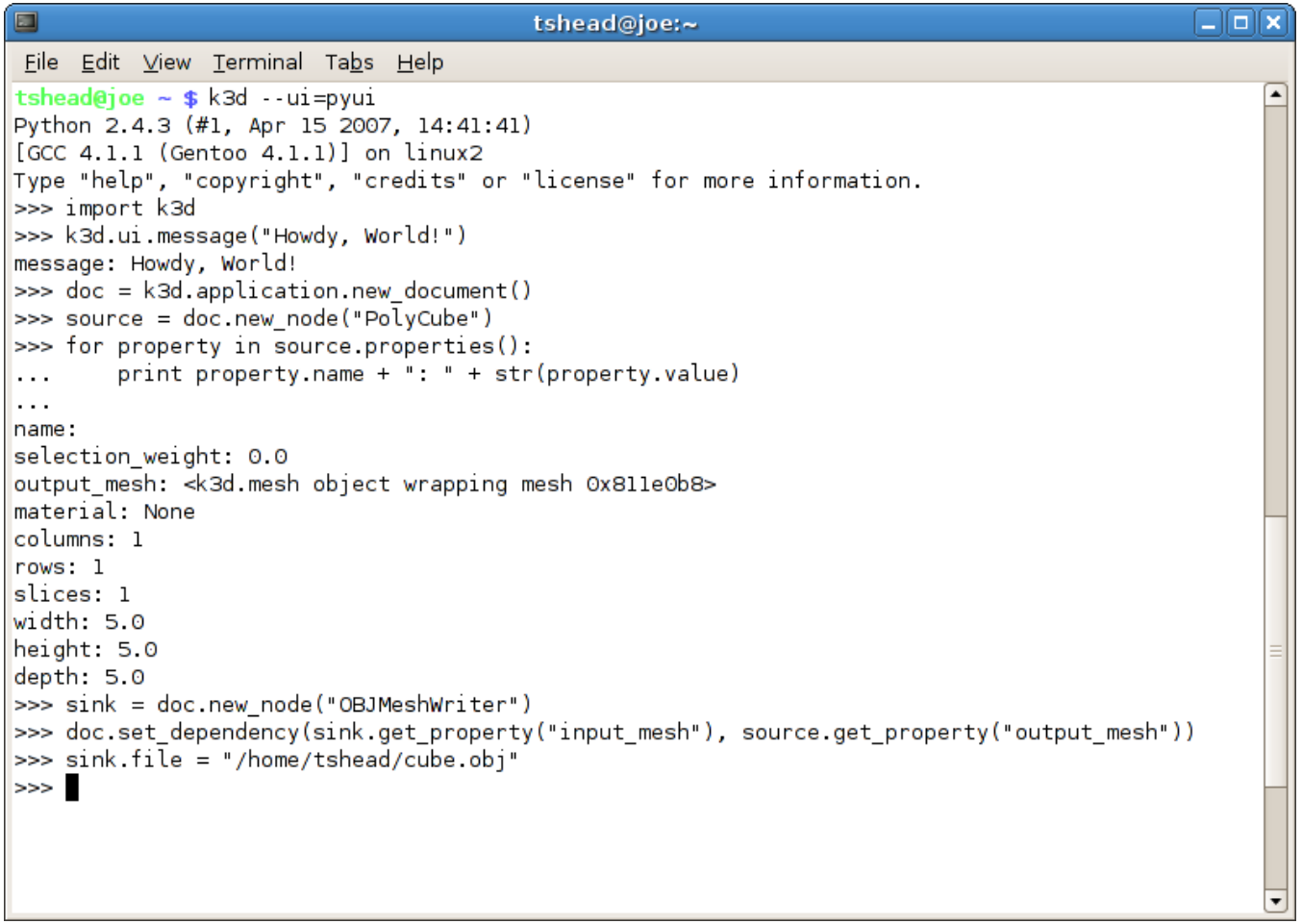
File Edit View Terminal Tabs Help

```
#python

import k3d

doc = k3d.new_document()

# Create a simple polyhedron source ...
source = doc.new_node("PolyCube")

# Select some geometry ...
select_face = doc.new_node("SelectFaceByNumber")
select_face.index = 1;
doc.set_dependency(select_face.get_property("input_mesh"), source.get_property("output_mesh"))

select_edge = doc.new_node("SelectEdgeByNumber")
select_edge.index = 2;
doc.set_dependency(select_edge.get_property("input_mesh"), select_face.get_property("output_mesh"))

select_point = doc.new_node("SelectPointByNumber")
select_point.index = 3;
doc.set_dependency(select_point.get_property("input_mesh"), select_edge.get_property("output_mesh"))

# Run the geometry through a conversion from mesh to legacy-mesh and back ...
convert = doc.new_node("LegacyMeshConversion")
doc.set_dependency(convert.get_property("input_mesh"), select_point.get_property("output_mesh"))

# Compare the original to the conversion ...
diff = doc.new_node("MeshDiff")
diff.add_user_property("k3d::dev::mesh*", "input_a", "InputA", "First input mesh")
diff.add_user_property("k3d::dev::mesh*", "input_b", "InputB", "Second input mesh")

doc.set_dependency(diff.get_property("input_a"), select_point.get_property("output_mesh"))
doc.set_dependency(diff.get_property("input_b"), convert.get_property("output_mesh"))

if not diff.equal:
        print "source " + repr(select_point.output_mesh)
        print "converted " + repr(convert.output_mesh)
        raise Exception("Converted mesh differs")
```

1,1 Top

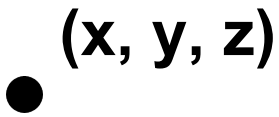
Sandia National Laboratories

Python command-line

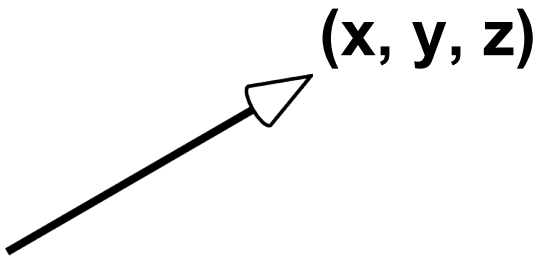


Sandia National Laboratories

Two common 3D graphics concepts



point3
+ x : double
+ y : double
+ z : double



vector3
+ x : double
+ y : double
+ z : double

Declaring point3 in C++

```
namespace k3d
{

class point3
{
public:
    point3();
    point3(double, double, double);

    double x, y, z;
};

}
```

Declaring vector3 in C++

```
namespace k3d
{

class vector3
{
public:
    vector3();
    vector3(double, double, double);

    double x, y, z;
};

}
```

“Wrapping” point3 and vector3 using Boost.Python

```
#include "point3.h"
#include "vector3.h"

#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(k3d)
{

class_<k3d::point3>("point3")
    .def(init<double, double, double>());

class_<k3d::vector3>("vector3")
    .def(init<double, double, double>());

}
```

Testing point3 and vector3 in Python

```
>>> import k3d

>>> dir(k3d)
['__doc__', '__name__', 'point3', 'vector3']

>>> p = k3d.point3(0, 1, 2)

>>> print p
<k3d.point3 object at 0xb7bb8b44>

>>> v = k3d.vector3(1, 2, 3)

>>> print v
<k3d.vector3 object at 0xb7bb8b94>
```

Declaring stream inserters in C++

```
#include <iosfwd>

namespace k3d
{

std::ostream& operator<<(std::ostream&,
    const point3&);

std::ostream& operator<<(std::ostream&,
    const vector3&);

} // namespace k3d
```

Wrapping stream inserters in Python

```
BOOST_PYTHON_MODULE(k3d)
{
class_<k3d::point3>("point3")
    .def(init<double, double, double>())
    .def(str(self));

class_<k3d::vector3>("vector3")
    .def(init<double, double, double>())
    .def(str(self));
}
```

Sandia National Laboratories

Testing stream inserters in Python

```
>>> print k3d.point3(0, 1, 2)
(0 1 2)

>>> print k3d.vector3(3, 4, 5)
(3 4 5)
```

Wrapping member data as Python properties

```
BOOST_PYTHON_MODULE(k3d)
{
class_<k3d::point3>("point3")
    .def(init<double, double, double>())
    .def_readwrite("x", &k3d::point3::x)
    .def_readwrite("y", &k3d::point3::y)
    .def_readwrite("z", &k3d::point3::z)
    .def(str(self));

class_<k3d::vector3>("vector3")
    .def(init<double, double, double>())
    .def_readwrite("x", &k3d::vector3::x)
    .def_readwrite("y", &k3d::vector3::y)
    .def_readwrite("z", &k3d::vector3::z)
    .def(str(self));
}
```

Testing Python properties

```
>>> p = k3d.point3(0, 1, 2)

>>> print p.x
0.0

>>> p.x = 2.5

>>> print p.x
2.5

>>> v = k3d.vector3(4, 5, 6)

>>> print v.y
5.0
```

Adding member methods in C++

```
class vector3
{
public:
    vector3();
    vector3(double, double, double);

    const vector3 normalized() const;

    double x, y, z;
};
```

```
#include <cmath>

const vector3 vector3::normalized() const
{
    const double length =
        std::sqrt(x * x + y * y + z * z);

    return vector3(
        x / length, y / length, z / length);
}
```

Wrapping member methods in Python

```
class_<k3d::vector3>("vector3")
    .def(init<double, double, double>())
    .def_readwrite("x", &k3d::vector3::x)
    .def_readwrite("y", &k3d::vector3::y)
    .def_readwrite("z", &k3d::vector3::z)
    .def("normalized", &k3d::vector3::normalized)
    .def(str(self));
```

```
>>> v = k3d.vector3(0, 1, 2)

>>> print v.normalized()
(0 0.447214 0.894427)
```

But there's a problem here ...

```
>>> v = k3d.vector3(0, 0, 0)

>>> print v.normalized()
(-1.#IND -1.#IND -1.#IND)
```

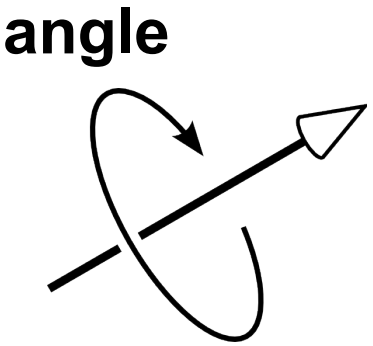
Throwing C++ exceptions

```
#include <stdexcept>

const vector3 vector3::normalized() const
{
    const double length =
        std::sqrt(x * x + y * y + z * z);

    if(!length)
        throw std::runtime_error(
            "zero-length vector");

    return vector3(
        x / length, y / length, z / length);
}
```

Automatic exception translation in Python

```
>>> v = k3d.vector3(0, 0, 0)

>>> print v.normalized()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: zero-length vector
```

Representing a rotation in 3D



angle_axis
+ angle : double
+ axis : vector3

Declaring angle_axis in C++

```
class angle_axis
{
public:
    angle_axis();
    angle_axis(double, const vector3&);

    double angle;
    vector3 axis;
};

std::ostream& operator<<(std::ostream&,
    const angle_axis&);
```

Wrapping angle_axis in Python

```
class_<k3d::angle_axis>("angle_axis")
    .def(init<double, const k3d::vector3&>())
    .def_readwrite("angle",
        &k3d::angle_axis::angle)
    .def_readwrite("axis",
        &k3d::angle_axis::axis)
    .def(str(self));
```

Demonstrating angle_axis in Python

```
>>> a = k3d.angle_axis(1.5708, k3d.vector3(0, 0, 1))

>>> print a
1.5708 (0 0 1)
```

Demonstrating angle_axis in Python

```
>>> a = k3d.angle_axis(1.5708, k3d.vector3(0, 0, 1))

>>> print a
1.5708 (0 0 1)
```

1.5708 radians ≈ 90°

Converting radians (intrusive)

```
class angle_axis
{
public:
    angle_axis(double, const vector3&);

    const double get_angle_degrees() const;
    void set_angle_degrees(double Angle);

    double angle;
    vector3 axis;
};
```

```
class_<k3d::angle_axis>("angle_axis")
    .def(init<double, const k3d::vector3&>())
    .add_property("angle",
        &k3d::angle_axis::get_angle_degrees,
        &k3d::angle_axis::set_angle_degrees)
    .def_readwrite("axis", &k3d::angle_axis::axis)
    .def(str(self));
```

It works as far as it goes ...

```
>>> a = k3d.angle_axis(1.5708, k3d.vector3(0, 0, 1))

>>> print a.angle
90.0000000003

>>> a.angle = 180

>>> print a
3.1415 (0 0 1)
```

Trick: Use non-member methods

In Boost.Python, you can substitute a non-member method for a member method, as long as it has an explicit "self" reference as the first argument:

```
class foo
{
public:
    void bar();
};
```

```
void foo_bar(foo& Self);
```

Note – just like you'd declare a class method in Python:

```
>>> class foo:
...     def bar(self):
```

A better (nonintrusive) solution

```
const double angle_axis_get_angle(
    const k3d::angle_axis& Self)
{
    return degrees(Self.angle);
}

void angle_axis_set_angle(
    k3d::angle_axis& Self,
    double Angle)
{
    Self.angle = radians(Angle);
}
```

Wrapping the nonintrusive version

```
class_<k3d::angle_axis>("angle_axis")
    .def(init<double, const k3d::vector3&>())
    .add_property("angle",
        &angle_axis_get_angle,
        &angle_axis_set_angle)
    .def_readwrite("axis", &k3d::angle_axis::axis)
    .def(str(self));
```

```
>>> a = k3d.angle_axis(1.5708, k3d.vector3(0, 0, 1))

>>> print a.angle
90.0000000003

>>> a.angle = 180

>>> print a
3.1415 (0 0 1) # Still displaying radians here!
```

Trick: You don't have to use Boost.Python's special methods

You can define any special Python method using the normal "def" syntax:

__str__	"Informal" string representation
__len__	Return container length
__getitem__	Return container item value
__setitem__	Set container item value
__hash__	Key for dictionary operations
__getattr__	Return attribute value
__setattr__	Set attribute value
__add__	Binary arithmetic
__mul__	Binary arithmetic
	... and many more!

Source: http://docs.python.org/ref/specialnames.html

Example: making point3 a container

```
const int point3_len(const k3d::point3&)
{
    return 3;
}
```

Making point3 a container

```
const double point3_getitem(
    const k3d::point3& Self,
    int Item)
{
    switch(Item)
    {
    case 0:
        return Self.x;
    case 1:
        return Self.y;
    case 2:
        return Self.z;
    default:
        throw std::out_of_range(
            "index out-of-range");
    }
}
```

Making point3 a container

```
void point3_setitem(
    k3d::point3& Self,
    int Item,
    double Value)
{
    switch(Item)
    {
    case 0:
        Self.x = Value; break;
    case 1:
        Self.y = Value; break;
    case 2:
        Self.z = Value; break;
    default:
        throw std::out_of_range(
            "index out-of-range");
    }
}
```

Making point3 a container

```
class_<k3d::point3>("point3")
    .def(init<double, double, double>())
    .def_readwrite("x", &k3d::point3::x)
    .def_readwrite("y", &k3d::point3::y)
    .def_readwrite("z", &k3d::point3::z)
    .def("__len__", &point3_len)
    .def("__getitem__", &point3_getitem)
    .def("__setitem__", &point3_setitem)
    .def(self == self)
    .def(str(self));
```

Container methods example

```
>>> p = k3d.point3(2, 4, 6)

>>> print len(p)
3

>>> print p[0]
2.0

>>> p[2] = 8

>>> for i in p:
...     print i
...
2.0
4.0
8.0
```

Replacing the “__str__” method

```
const std::string angle_axis_str(
    const k3d::angle_axis& Self)
{
    std::stringstream buffer;
    buffer
        << degrees(Self.angle)
        << " " << Self.axis;
    return buffer.str();
}

class_<k3d::angle_axis>("angle_axis", no_init)
    .add_property("angle",
        &angle_axis_get_angle,
        &angle_axis_set_angle)
    .def_readwrite("axis", &k3d::angle_axis::axis)
    .def("__str__", &angle_axis_str);
```

Custom “__str__” example

```
>>> a = k3d.angle_axis(1.5708, k3d.vector3(0, 0, 1))

>>> print a.angle
90.0000000003

>>> print a
90 (0 0 1)
```

Constructor workaround (1st try)

```
const k3d::angle_axis construct_angle_axis(
    double Angle,
    const k3d::vector3& Axis)
{
    return k3d::angle_axis(radians(Angle), Axis);
}

BOOST_PYTHON_MODULE(k3d)
{
def("angle_axis", &construct_angle_axis);

class_<k3d::angle_axis>("angle_axis", no_init)
    .add_property("angle",
        &angle_axis_get_angle,
        &angle_axis_set_angle)
    .def_readwrite("axis", &k3d::angle_axis::axis)
    .def(str(self));
}
```

Constructor workaround demo

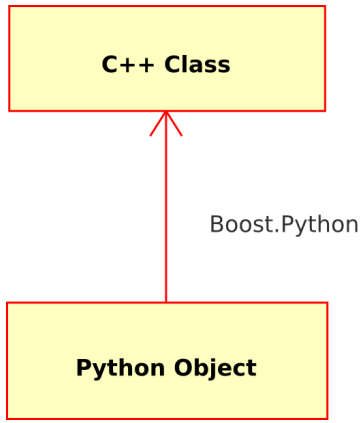
```
>>> a = k3d.angle_axis(90, k3d.vector3(0, 0, 1))

>>> print a.angle
90.0000000003

>>> print a
90 (0 0 1)
```

It works, but it's dissatisfying:

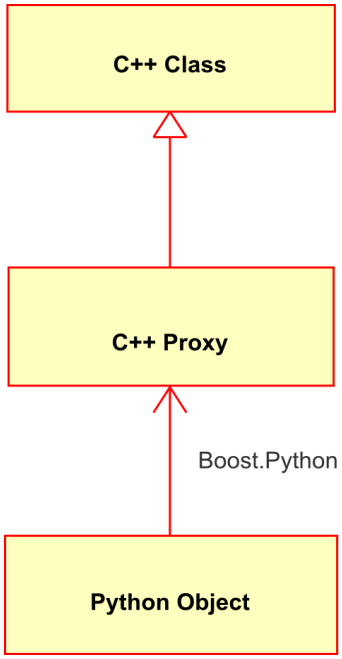
- **It bypasses Boost.Python's syntax for overloaded constructors.**
- **Declaring a class and a method with the same name causes confusion.**

Trick: Use “proxy” classes

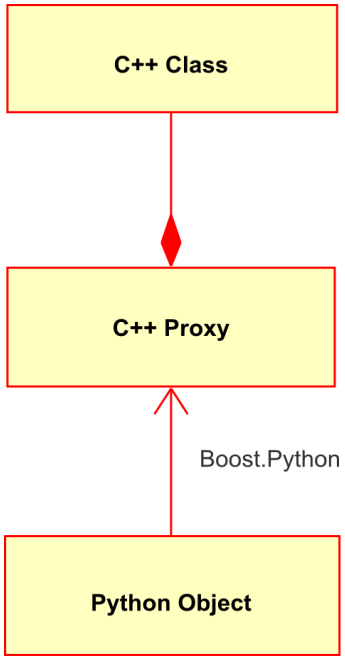
- **Proxies add the “extra level of indirection” that solves difficult problems.**
- **Proxies provide better code organization.**



w/o proxy



“is-a”



“has-a”

Sandia National Laboratories

Declaring angle_axis_proxy

```
class angle_axis_proxy :
    public k3d::angle_axis
{
    typedef k3d::angle_axis base;

public:
    angle_axis_proxy();
    angle_axis_proxy(
        const k3d::angle_axis& Value);
    angle_axis_proxy(
        double Angle,
        const vector3& Axis);

    const double get_angle() const;
    void set_angle(const double Value);
    const std::string str() const;

    static void define_python_class();
};
```

Defining angle_axis_proxy

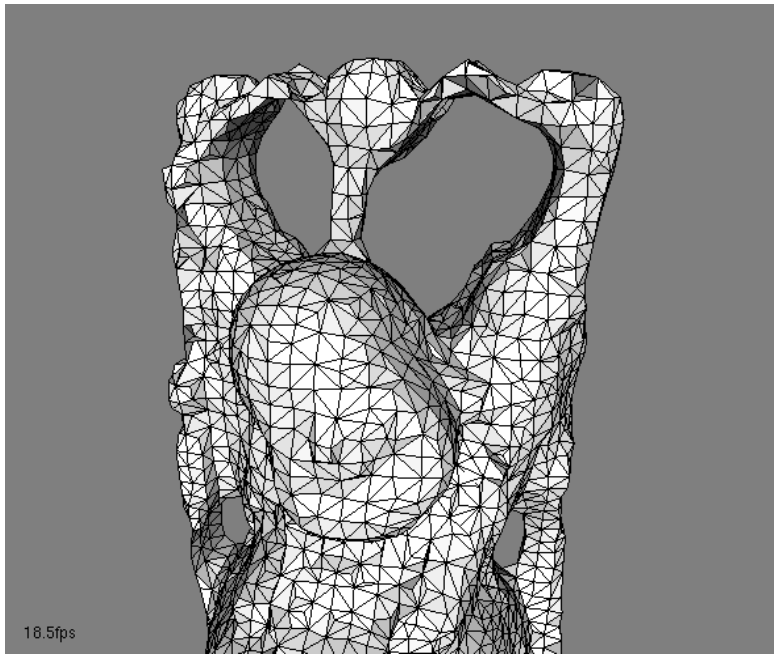
```
angle_axis_proxy::angle_axis_proxy(
    double Angle,
    const vector3& Axis) :
    base(radians(Angle), Axis)
{
}

/* other methods here */

void angle_axis_proxy::define_python_class()
{
    class_<angle_axis_proxy>("angle_axis",
        .def(init<double, const k3d::vector3&>())
        .add_property("angle",
            &angle_axis_proxy::get_angle,
            &angle_axis_proxy::set_angle)
        .def_readwrite("axis",
            &angle_axis_proxy::axis)
        .def("__str__", &angle_axis_proxy::str);
}
```

Wrapping angle_axis_proxy

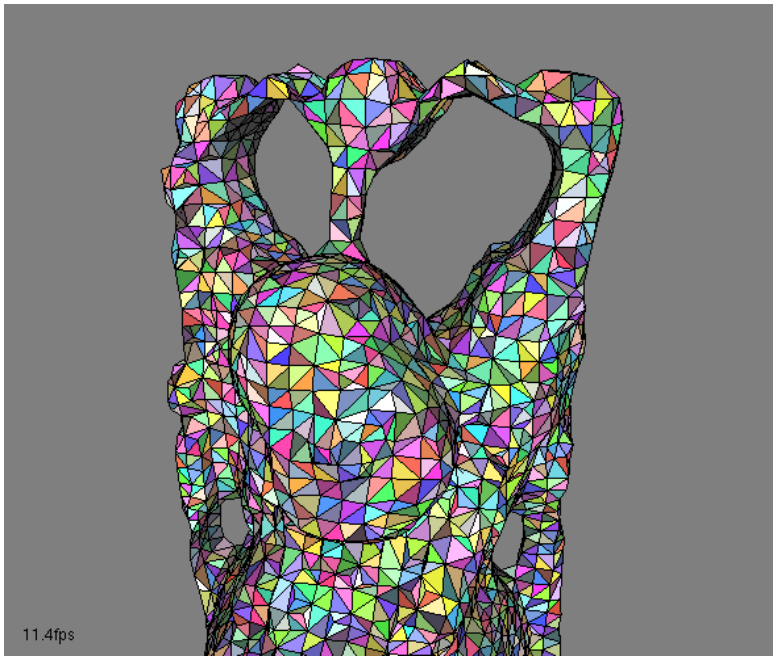
```
BOOST_PYTHON_MODULE(k3d)
{
    angle_axis_proxy::define_python_class();
}
```

```
>>> a = k3d.angle_axis(90, k3d.vector3(0, 0, 1))

>>> print a.angle
90.0000000003

>>> print a
90 (0 0 1)
```

Mesh datastructures



vertex + connectivity



vertex + connectivity + per-face-color

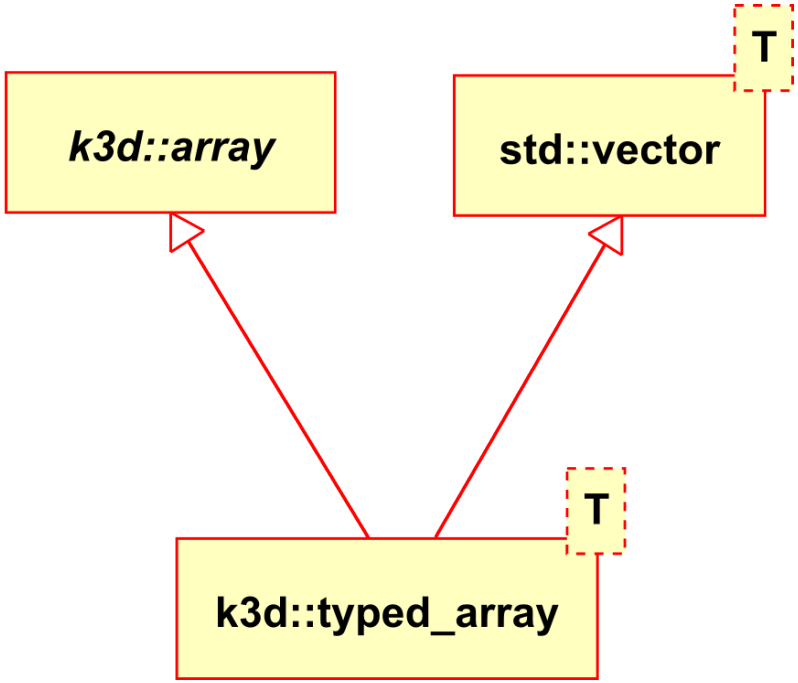
Random per-face color script

random_per_face_colors.py

File Edit

```
#python

import k3d
from random import seed, uniform

Output.copy(Input)

seed(123)

if Output.bilinear_patches() and Output.bilinear_patches().patch_points():
	Cs = Output.writable_bilinear_patches().writable_uniform_data().create_array("Cs", "k3d::color")
	for i in range(len(Output.bilinear_patches().patch_points()) / 4):
		Cs.append(k3d.color(uniform(0, 1), uniform(0, 1), uniform(0, 1)))

if Output.bicubic_patches() and Output.bicubic_patches().patch_points():
	Cs = Output.writable_bicubic_patches().writable_uniform_data().create_array("Cs", "k3d::color")
	for i in range(len(Output.bicubic_patches().patch_points()) / 16):
		Cs.append(k3d.color(uniform(0, 1), uniform(0, 1), uniform(0, 1)))

if Output.nurbs_patches() and Output.nurbs_patches().patch_first_points():
	Cs = Output.writable_nurbs_patches().writable_uniform_data().create_array("Cs", "k3d::color")
	for i in range(len(Output.nurbs_patches().patch_first_points())):
		Cs.append(k3d.color(uniform(0, 1), uniform(0, 1), uniform(0, 1)))

if Output.polyhedra() and Output.polyhedra().face_first_loops():
	Cs = Output.writable_polyhedra().writable_uniform_data().create_array("Cs", "k3d::color")
	for i in range(len(Output.polyhedra().face_first_loops())):
		Cs.append(k3d.color(uniform(0, 1), uniform(0, 1), uniform(0, 1)))
```

Heterogeneous array design



Array declarations in C++

```
#include <vector>

class array
{
public:
    virtual ~array() {}
};

template<typename T>
class typed_array :
    public std::vector<T>,
    public array
{
public:
    /* Provide std::vector<> -like ctors here */
};

typedef typed_array<point3> point3_array;
typedef typed_array<vector3> vector3_array;
```

Wrapping point3_array

```
class point3_array_proxy :
    public k3d::point3_array
{
    typedef k3d::point3_array base;
public:
    const int len() const;
    const point3 getitem(int Item) const;
    void setitem(int Item, const point3& Value);

    static void define_python_class();
};

BOOST_PYTHON_MODULE(k3d)
{
    point3_array_proxy::define_python_class();
}
```

Wrapping vector3_array

```
class vector3_array_proxy :
    public k3d::vector3_array
{
    typedef k3d::vector3_array base;
public:
    const int len() const;
    const vector3 getitem(int Item) const;
    void setitem(int Item, const vector3& Value);

    static void define_python_class();
};

BOOST_PYTHON_MODULE(k3d)
{
    vector3_array_proxy::define_python_class();
}
```

Trick: You can use templates with Boost.Python, as long as they're specialized

You can't map a template function/class into Python directly, but you can use a fully-specialized function/class anywhere you like.

The main caveat: naming

Templated array_proxy declaration

```
template<class ArrayT>
class array_proxy
{
public:
    const int len() const;
    const typename ArrayT::value_type getitem(
        int Item) const;
    void setitem(
        int Item,
        const typename ArrayT::value_type& Value);
    const std::string str() const;

    static void define_python_class(
        const char* ClassName);

private:
    ArrayT array;
};
```

Templated array_proxy definition

```
template<class ArrayT>
const int array_proxy<ArrayT>::len() const
{
    return array.size();
}

template<class ArrayT>
const typename ArrayT::value_type
array_proxy<ArrayT>::getitem(int Item) const
{
    if(Item < 0 || Item >= array.size())
        throw std::out_of_range(
            "index out-of-range");

    return array[Item];
}
```

Templated array_proxy definition

```
template<class ArrayT>
void array_proxy<ArrayT>::setitem(int Item, const
typename ArrayT::value_type& Value)
{
    if(Item < 0)
        throw std::out_of_range(
            "index out-of-range");

    if(Item >= array.size())
        array.resize(Item + 1);

    array[Item] = Value;
}
```

Templated array_proxy definition

```
template<class ArrayT>
const std::string array_proxy<ArrayT>::str() const
{
    std::ostringstream buffer;

    std::copy(
        array.begin(),
        array.end(),
        std::ostream_iterator
            <typename ArrayT::value_type>
            (buffer, " "));

    return buffer.str();
}
```

Templated array_proxy definition

```
template<class ArrayT>
void array_proxy<ArrayT>::define_python_class(
    const char* ClassName)
{
    class_<array_proxy<ArrayT> >(ClassName)
        .def("__len__", &array_proxy<ArrayT>::len)
        .def("__getitem__",
            &array_proxy<ArrayT>::getitem)
        .def("__setitem__",
            &array_proxy<ArrayT>::setitem)
        .def("__str__", &array_proxy<ArrayT>::str);
}
```

Specializing array_proxy

```
BOOST_PYTHON_MODULE(k3d)
{

array_proxy<k3d::point3_array>::define_python_class(
    "point3_array");

array_proxy<k3d::vector3_array>::define_python_class(
    "vector3_array");

}
```

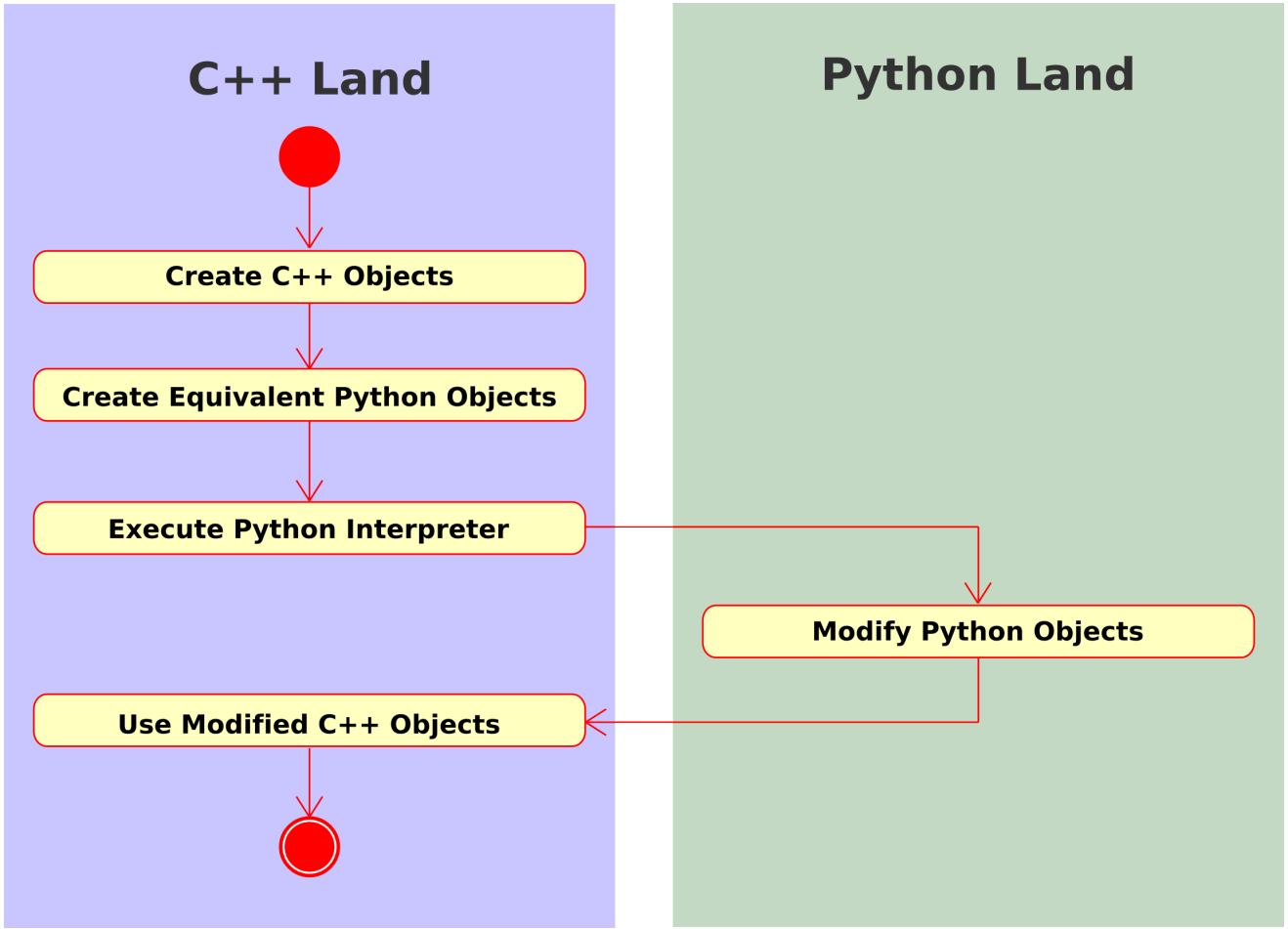
Demonstrating array_proxy

```
>>> a = k3d.point3_array()

>>> print len(a)
0

>>> a[0] = k3d.point3(0, 1, 2)

>>> a[1] = k3d.point3(3, 4, 5)

>>> print len(a)
2

>>> print a
(0 1 2) (3 4 5)
```

Demonstrating array_proxy

```
>>> a = k3d.vector3_array()

>>> print len(a)
0

>>> a[0] = k3d.vector3(0, 0, 1)

>>> a[1] = k3d.vector3(0, 1, 2)

>>> print len(a)
2

>>> print a
(0 0 1) (0 1 2)
```

Embedding use-case: extend C++ application functionality with Python



Three problems to solve:

1) Initialization: in the embedded use-case we have to handle some additional initialization ourselves.

2) Object lifetimes: we need to decouple the lifetimes of C++ and Python objects.

3) Instantiation: we need to instantiate Python objects from our C++ code.

Trick: Embedded initialization

- Write normal Boost.Python code.
- Link the resulting library into your executable.
- Always call Py_Initialize() first thing!
- Call the magic entry point for your Boost.Python code.

Initialization example

```
BOOST_PYTHON_MODULE(k3d)
{
    // Good stuff here
}
```

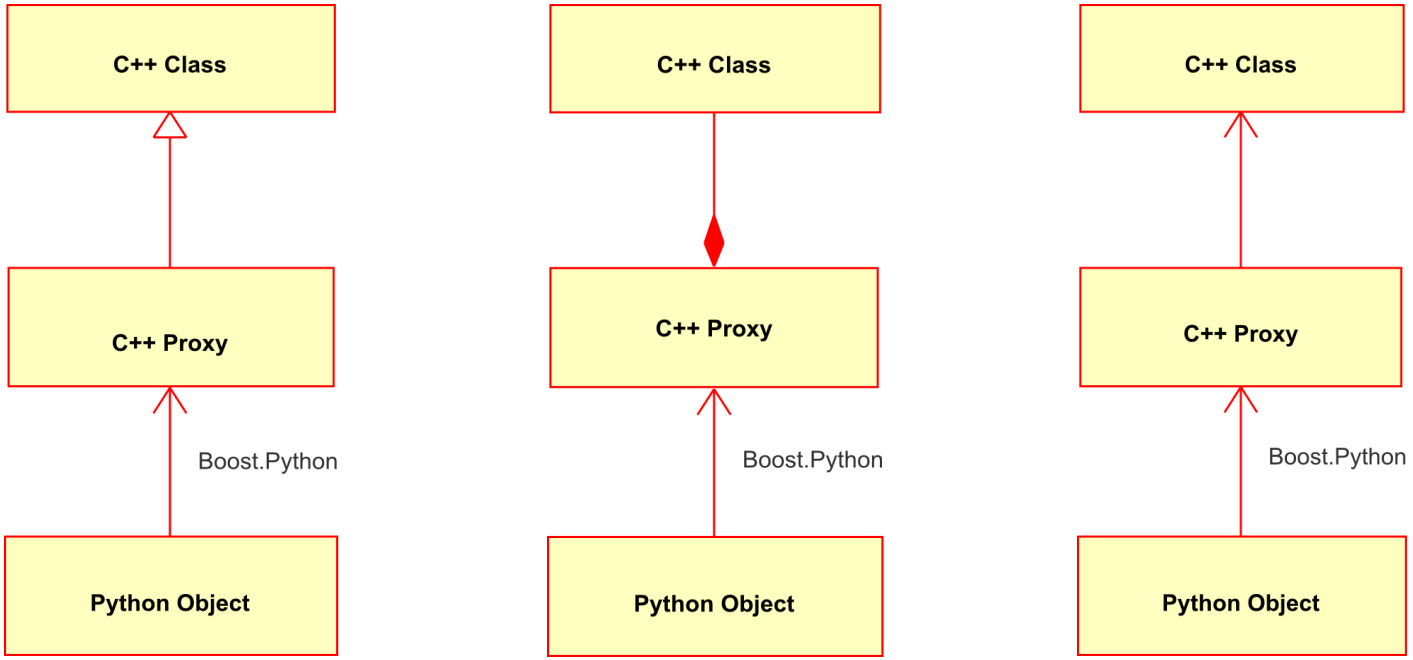
```
#include <Python.h>

extern "C" { void initk3d(); }

int main(int argc, char* argv[])
{
    Py_Initialize();

    initk3d();

    return Py_Main(argc, argv);
}
```

A more realistic example

```
void run_embedded_script(
    const std::string& Script)
{
    if(!Py_IsInitialized())
    {
        Py_Initialize();
        initk3d();
    }

    PyRun_SimpleString(Script.c_str());
}
```

Trick: “Proxy” classes redux

Proxies can also be used to decouple the lifetimes of C++ and Python objects using the “handle-body” idiom.



“is-a” “has-a” “refers-to”

Updated array_proxy declaration

```
template<class ArrayT>
class array_proxy
{
public:
    array_proxy(ArrayT& Array);

    /* ... */

private:
    ArrayT& array;
};
```

Updated array_proxy definition

```
template<class ArrayT>
array_proxy<ArrayT>::array_proxy(ArrayT& Array) :
    array(Array)
{
}

template<class ArrayT>
void array_proxy<ArrayT>::define_python_class(
    const char* ClassName)
{
    class_<array_proxy<ArrayT> >(ClassName, no_init)

    /* other stuff here */
}
```

Updated array_proxy demonstration

```
>>> a = k3d.point3_array()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: This class cannot be instantiated
from Python

>>> a = k3d.vector3_array()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: This class cannot be instantiated
from Python
```

Trick: Reflecting C++ objects into Python

- Wrap any C++ object into Python using boost::python::object.
- Put the boost::object instances into a dict
  - boost::python::dict is convenient.
- Pass the dict to the Python interpreter and its contents become part of the Python environment.

Wrap C++ objects using boost::python::object

```
k3d::point3_array points = /* whatever */
k3d::vector3_array normals = /* whatever */

array_proxy<k3d::point3_array>
    points_proxy(points);
array_proxy<k3d::vector3_array>
    normals_proxy(normals);

boost::python::object
    points_object(points_proxy);
boost::python::object
    normals_object(normals_proxy);
```

Put the boost::python::object instances into a dict

```
boost::python::dict local_dict;

local_dict["Points"] = points_object;
local_dict["Normals"] = normals_object;
```

Pass the dict to the Python interpreter

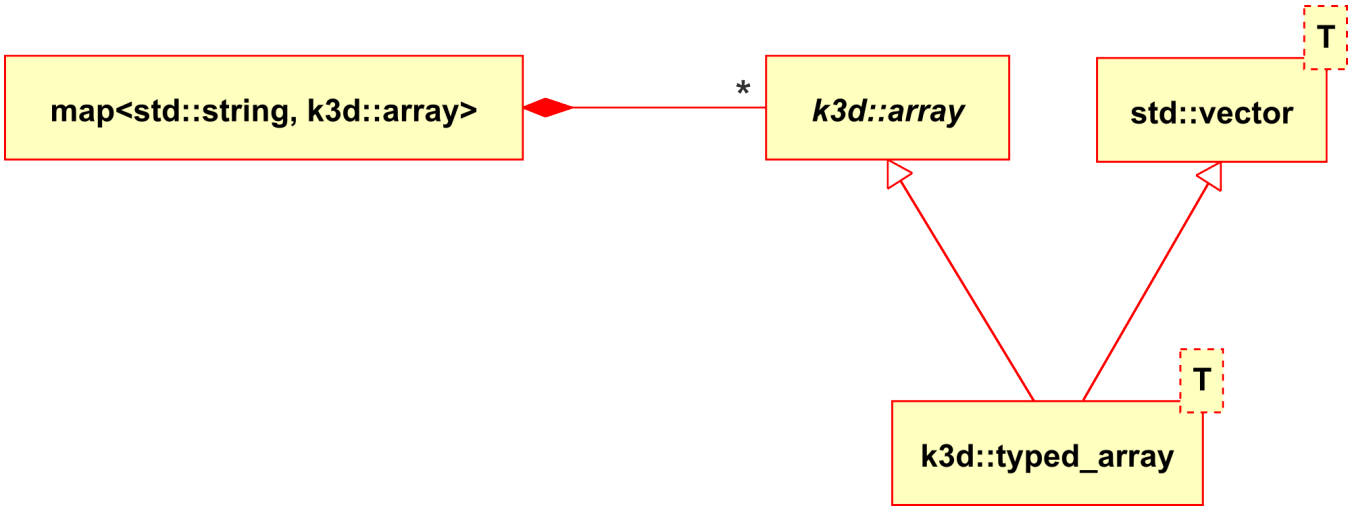
```
PyRun_String(
    const_cast<char*>(script.c_str()),
    Py_file_input,
    local_dict.ptr(),
    local_dict.ptr());
```

Thanks go to Gerhard Reitmayr for the correct scoping of dictionaries in Python

Sample script

```
# Translate every point along the X axis ...
for point in Points:
        points.x += 5

# Ensure all normals have length one ...
for normal in Normals:
        normal = normal.normalized()
```

Expanded heterogeneous array design




```
typedef std::map<std::string, array*> named_arrays;
```

Trick: Use boost::python::object for weakly-typed arguments and results

For a function that can return more than one type (or None):

```
boost::python::object my_function();
```

For a function that can accept more than one type (or None) as an argument:

```
void my_function(boost::python::object Argument);
```

Declaring named_arrays_proxy

```
class named_arrays_proxy
{
public:
    named_arrays_proxy(k3d::named_arrays& Arrays);

    boost::python::object get_array(
        const std::string& Name);

    boost::python::object create_array(
        const std::string& Name,
        const std::string& Type);

    void delete_array(
        boost::python::object& Array);

    static void define_python_class();

private:
    k3d::named_arrays& arrays;
};
```

Defining named_arrays_proxy

```
boost::python::object named_arrays_proxy::get_array(
    const std::string& Name)
{
    if(0 == arrays.count(Name))
        return boost::python::object();

    k3d::array* const abstract_array = arrays[Name];

    if(k3d::point3_array* const concrete_array =
        dynamic_cast<k3d::point3_array*>(abstract_array))
    {
        return boost::python::object(
            array_proxy<k3d::point3_array>(*concrete_array));
    }

    /* Handle k3d::vector3_array and other array types here */

    throw std::runtime_error("Unknown array type");
}
```

Defining named_arrays_proxy

```
boost::python::object named_arrays_proxy::create_array(
    const std::string& Name,
    const std::string& Type)
{
    if(arrays.count(Name))
        throw std::invalid_argument(
            "array already exists");

    if(Type == "point3")
    {
        arrays[Name] = new k3d::point3_array();
        return boost::python::object(
            array_proxy<k3d::point3_array>(*arrays[Name]));
    }

    /* Handle k3d::vector3_array and other array types here */

    throw std::invalid_argument(
        "unknown array type");
}
```

Defining named_arrays_proxy

```
void named_arrays_proxy::delete_array(
    boost::python::object Array)
{
    boost::python::extract<std::string> array_name(Array);
    if(array_name.check())
    {
        delete_array_by_name(array_name());
    }

    boost::python::extract<array_proxy<k3d::point3_array> >
        point3_array(Array);
    if(point3_array.check())
    {
        delete_point3_array(point3_array());
    }

    /* Handle k3d::vector3_array and other array types here */

    throw std::invalid_argument("unknown array");
}
```

Reflecting named_arrays_proxy into the Python environment

```
k3d::named_arrays arrays;
arrays["Points"] = new k3d::point3_array();

named_arrays_proxy arrays_proxy(arrays);

boost::python::dict local_dict;

local_dict["Arrays"] =
    boost::python::object(arrays_proxy);

PyRun_String(
    const_cast<char*>(script.c_str()),
    Py_file_input,
    local_dict.ptr(),
    local_dict.ptr());
```

Testing named_arrays_proxy

```
>>> print Arrays.get_array("Normals")
None

>>> Normals = Arrays.create_array("Normals", "vector3")

>>> Normals[0] = k3d.normal3(0, 0, 1)
>>> Normals[1] = k3d.normal3(0, 1, 2)

>>> print Normals
(0 0 1)
(0 1 2)

>>> Arrays.delete("Points")

>>> Arrays.delete(Normals)
```

Summary

- Python objects can contain any combination of C++ class methods and non-member methods.
- “Special” Python methods can be implemented as normal member methods in Boost.Python.
- “Proxy” classes can be used to “override” class methods, improve code organization and decouple the lifetimes of C++ and Python objects.
- C++ templates can be wrapped using Boost.Python, as long as they're fully specialized.
- Embedding Boost.Python modules requires some additional initialization.
- boost::python::object can be used to implement methods in C++ that return / accept unrelated types.

Bonus: Additional proxy ideas

- Use proxies to map multiple C++ classes into a single Python class.
- Map a single C++ class into multiple Python classes.
- Add Python-specific functionality that doesn't exist in the C++ code (convenience methods, etc).

Questions?

Sandia National Laboratories