3D Data Visualization with VTK

Philly Codefest Workshop

Joel Pepper

PhD Student, Advisor: David Breen, Email: jcp353 [AT] drexel [DOT] edu

Drexel University

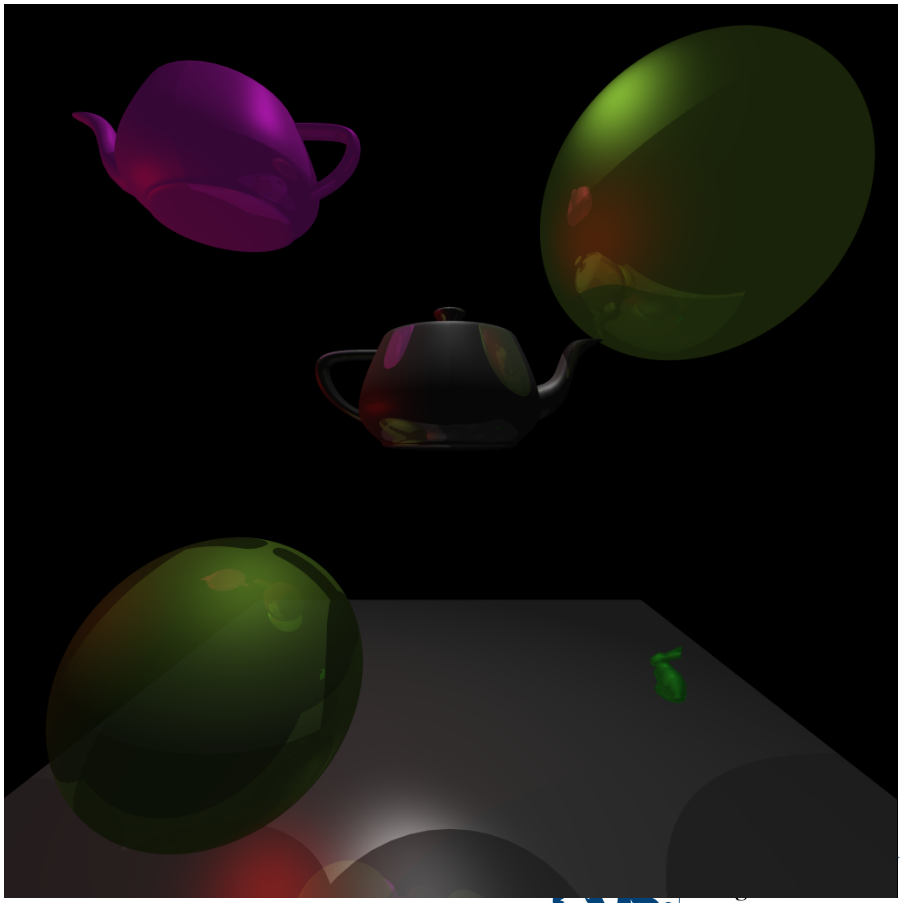
March 15, 2022

Outline

DREXEL UNIVERSITY College of Computing & Informatics

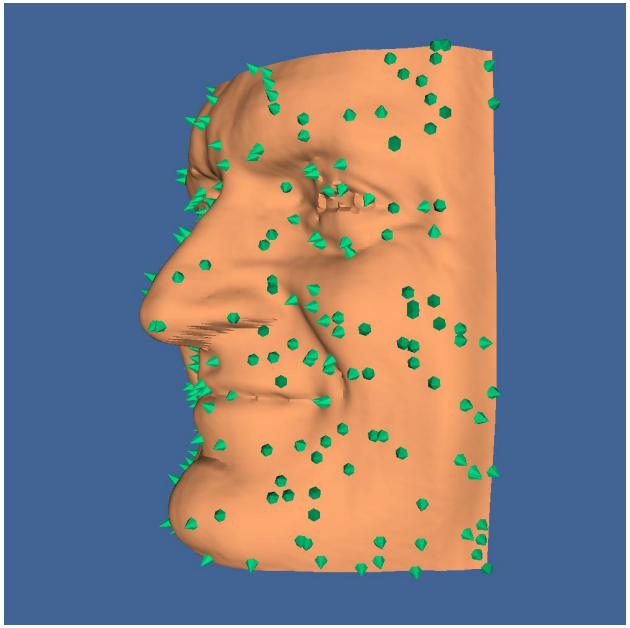
Graphics Introduction

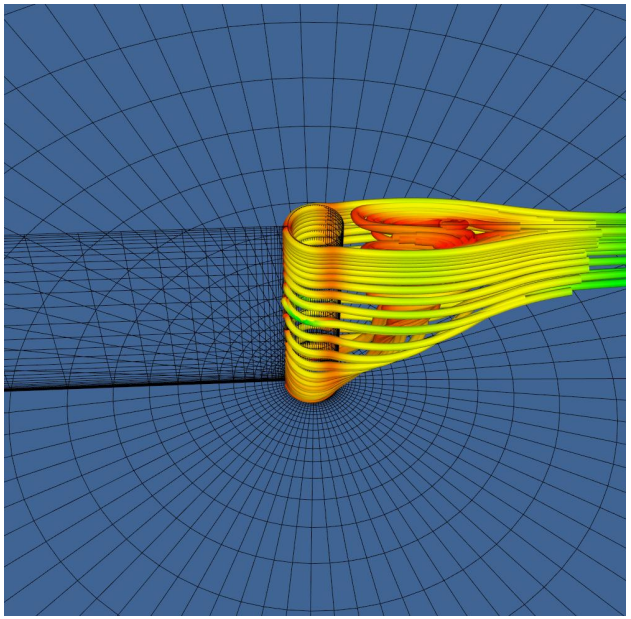
- 3D graphics rendering is complex, tedious and slow if you try to do it from scratch
- Even with GPU's and things like OpenGL, DirectX or (especially) Vulkan, it still constitutes very low level systems programming with a lot of overhead
- For tasks like data visualization, we want a higher level toolkit to simplify and accelerate the development process:
  Enter VTK

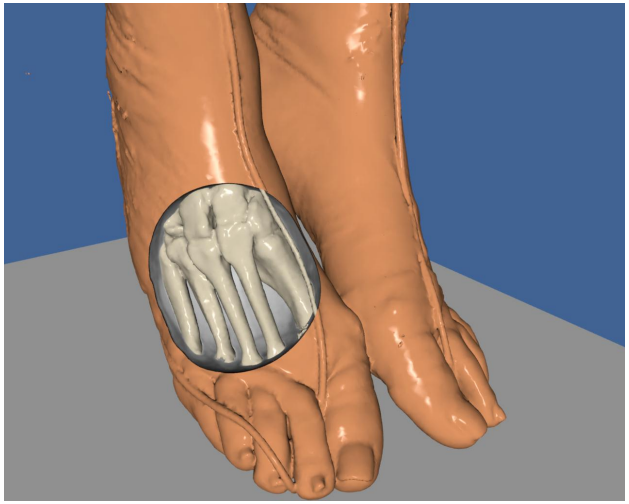


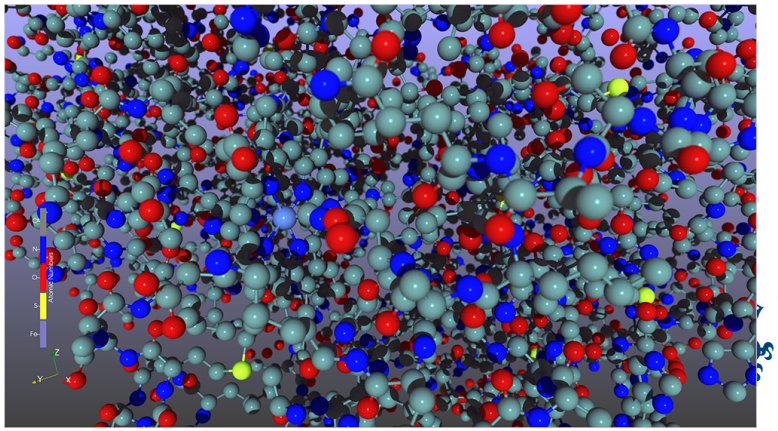
Computing & Informatics

VTK Official Examples







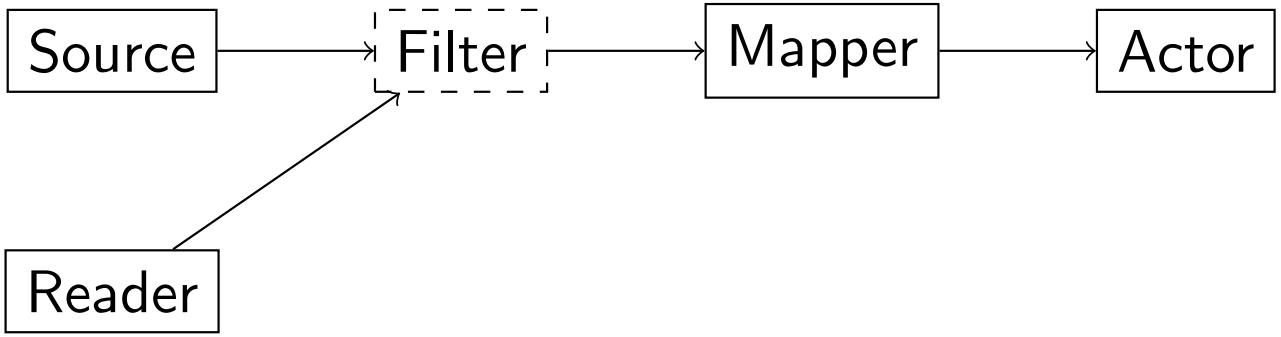


DREXEL UNIVERSITY
College of
Computing & Informatics

https://vtk.org/vtk-in-action/

VTK Workflow

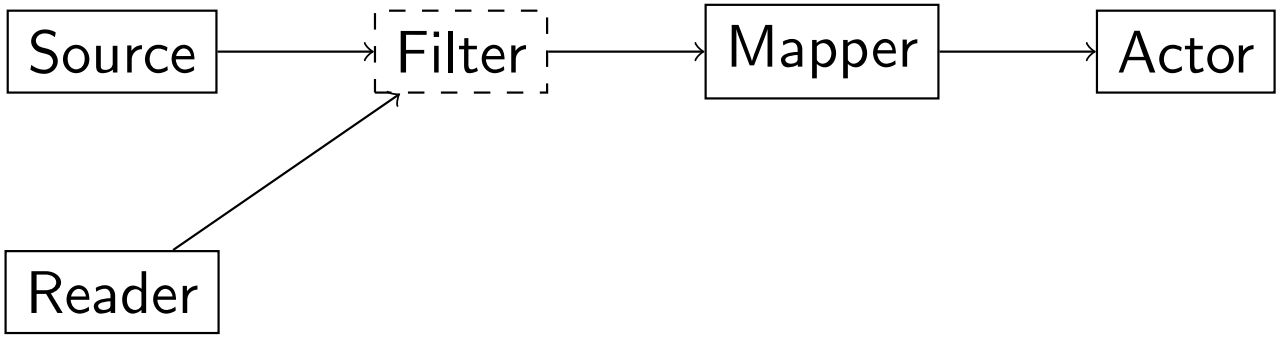
- Demo code: https://github.com/DrJPepper/codefest22-vtk-demo
- VTK programs consist of a scene of actors within a renderer within a render window
- Actors can be thing like lines, curves, cubes, spheres and triangle meshes
- Actors are created via the following workflow [1]:



- **Source**: Used to generate the geometry and topology of well defined shapes

VTK Workflow

- Demo code: https://github.com/DrJPepper/codefest22-vtk-demo
- VTK programs consist of a scene of actors within a renderer within a render window
- Actors can be thing like lines, curves, cubes, spheres and triangle meshes
- Actors are created via the following workflow [1]:

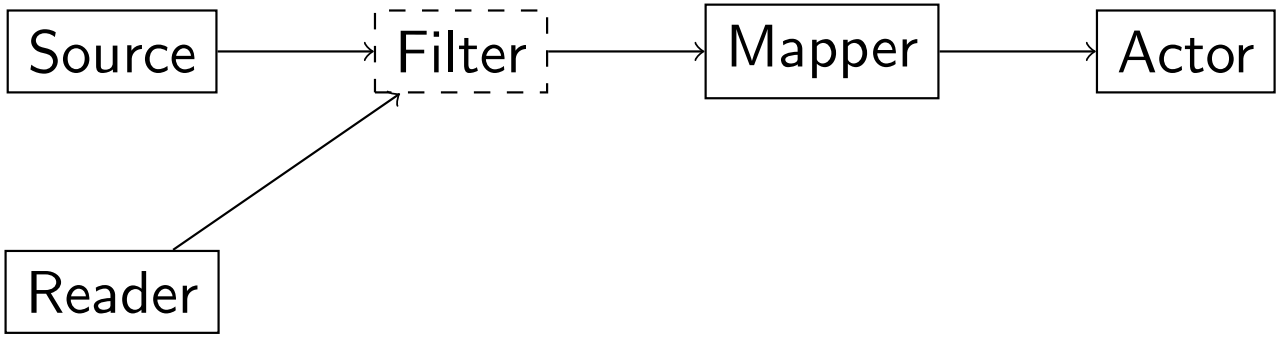


- **Reader**: Used to read geometry and topology from CAD files

UNIVERSITY of Computing & Informatics

VTK Workflow

- Demo code: https://github.com/DrJPepper/codefest22-vtk-demo
- VTK programs consist of a scene of actors within a renderer within a render window
- Actors can be thing like lines, curves, cubes, spheres and triangle meshes
- Actors are created via the following workflow [1]:

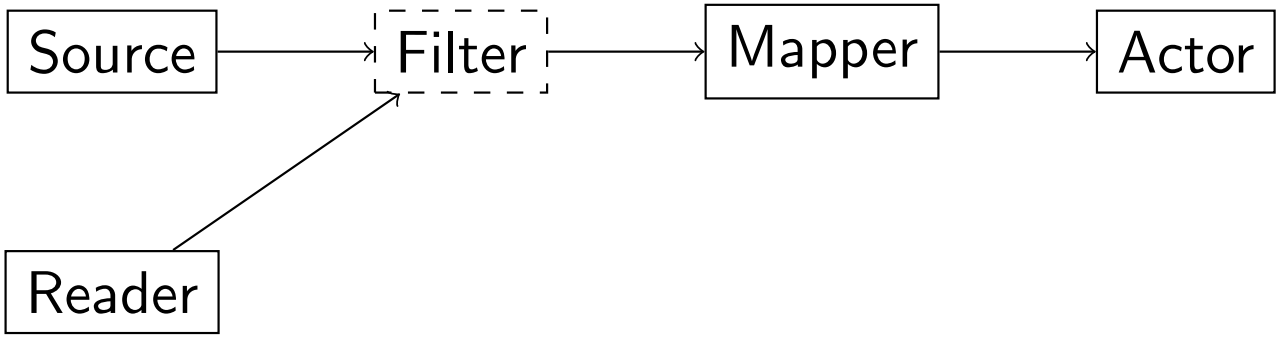


- **Filter**: Used to mutate data objects (not always explicitly needed)

UNIVERSITY of Computing & Informatics

VTK Workflow

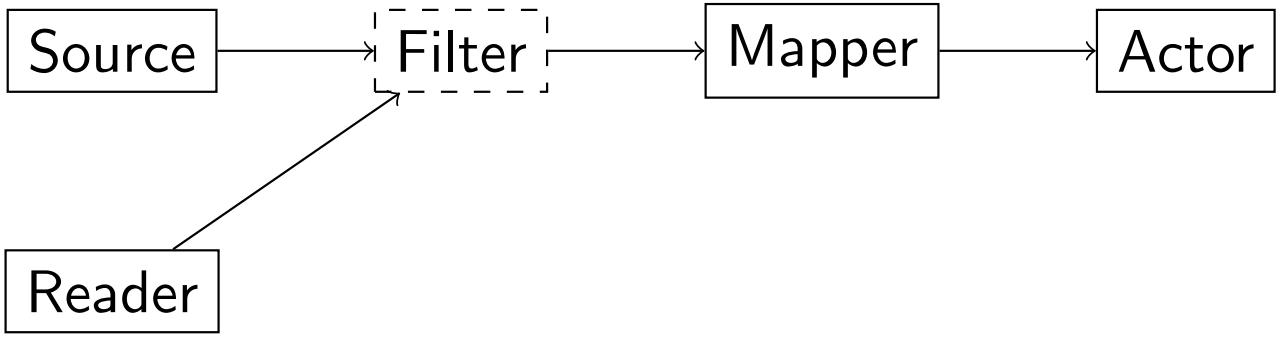
- Demo code: https://github.com/DrJPepper/codefest22-vtk-demo
- VTK programs consist of a scene of actors within a renderer within a render window
- Actors can be thing like lines, curves, cubes, spheres and triangle meshes
- Actors are created via the following workflow [1]:



- **Mapper**: Converts from basic geometry definitions to renderings

VTK Workflow

- Demo code: https://github.com/DrJPepper/codefest22-vtk-demo
- VTK programs consist of a scene of actors within a renderer within a render window
- Actors can be thing like lines, curves, cubes, spheres and triangle meshes
- Actors are created via the following workflow [1]:



- **Actor**: Renderable objects that contains all geometry, topology and visual properties needed to draw something in the scene

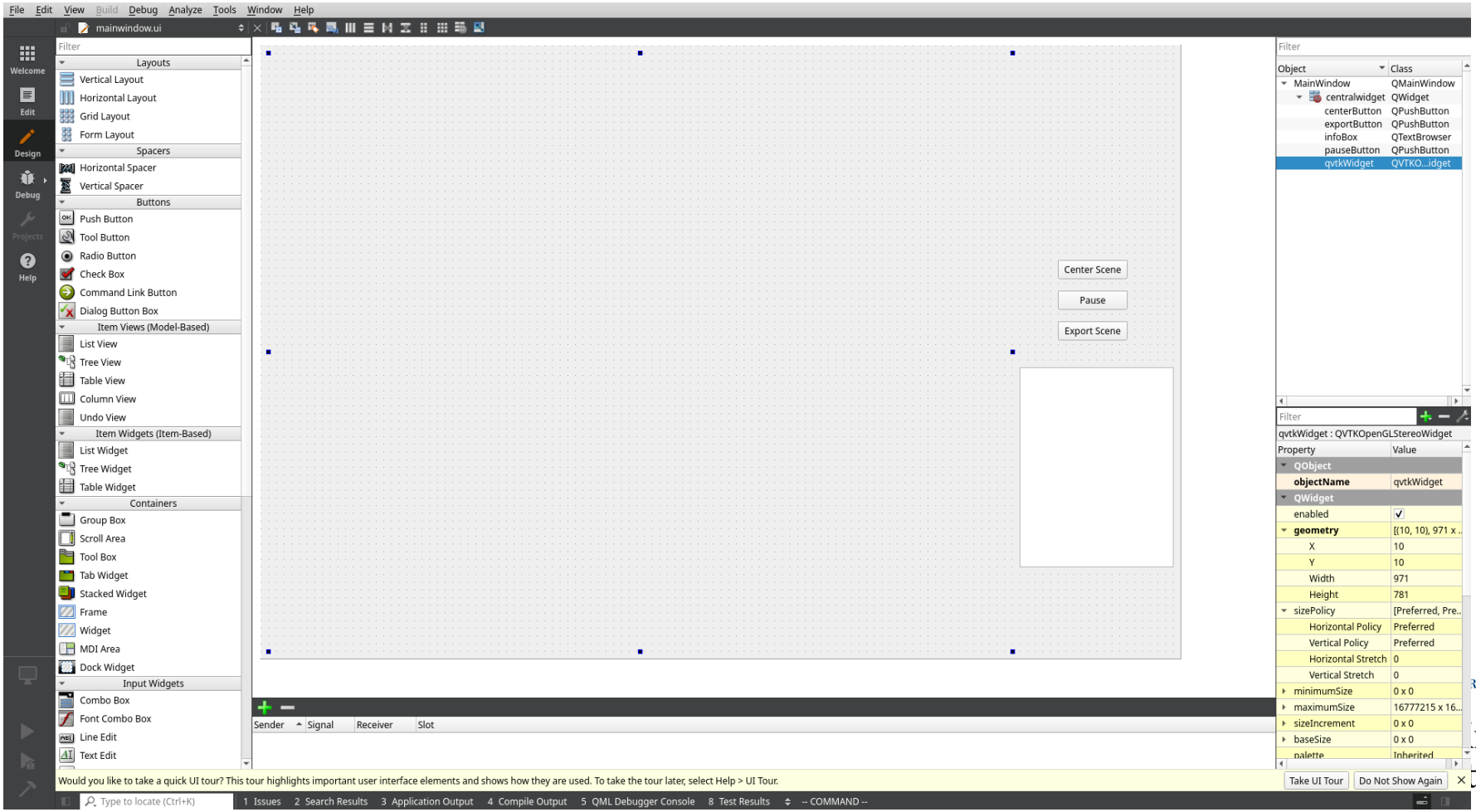
Application Frameworks

- VTK includes things like buttons, sliders, input fields and text boxes via what it refers to as "Widgets", and can be used standalone to create fully fledged GUI programs
- Widgets are rendered on top of a VTK scene however, and their configuration can be clunky
- In order to more easily create traditional desktop applications, you want to use an application framework such as GTK or Qt
- In this talk I will be covering a very basic introduction to Qt

DREXEL UNIVERSITY
College of
Computing &
Informatics

Qt Basics

- The easiest way to use Qt is to create `.ui` files with a program called `qtcreator`



Qt Basics

- Qt is based heavily around something called the "Meta-object systems" or MOC
- This is a framework and executable (`moc`) that takes special Qt enabled C++ programs as input, and convert them to valid C++ code with additional Qt features
- In addition to processing `.ui` files into C++ code, MOC also enables signals and slots for inter object communication
  - For example, a button press can trigger a function to run, or a (non blocking) timer can call a specific function every N milliseconds

MOC Code

mainwindow.h:

```
class MainWindow : public QMainWindow {
    Q_OBJECT

    public:
        MainWindow(QWidget *parent = 0);
        virtual ~MainWindow();
        ...

    private Q_SLOTS:
        void functionTimerCalls();
        void on_pauseButton_clicked();
        void on_exportButton_clicked();
        void on_centerButton_clicked();

    private:
        ...
};
```

DREXEL UNIVERSITY
College of
Computing &
Informatics

Main Qt Runner Code

main.cpp:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWindow;
    mainWindow.show();
    return app.exec();
}
```

DREXEL UNIVERSITY
College of
Computing &
Informatics

Side Note: CMake

- I don't have time to delve into this very deeply, but building a C++ project with these complex libraries more or less requires an automated build system of some kind
- I (and most other FOSS devoted people) use CMake, which automates finding and configuring libraries and project files for the compiler, assembler, linker and also calling the `moc` program

Side Note: CMake [2]

```
cmake_minimum_required(VERSION 3.16)

# Add folder where supporting functions are
set(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake)
set(CMAKE_INCLUDE_CURRENT_DIR ON)

if(NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Debug)
endif()

# Build settings
set(CMAKE_CXX_FLAGS "-Wall -O2 -fcompare-debug-second")
set(CMAKE_CXX_FLAGS_DEBUG "-ggdb")
set_property(TARGET ${PROJECT_NAME} PROPERTY CXX_STANDARD 20)

# Include basic Qt functions
include(QtCommon)

project(joelsvtkdemo VERSION 1.0)

# Set PROJECT_VERSION_PATCH and PROJECT_VERSION_TWEAK to 0 if not present, needed by add_project_meta
fix_project_version()

# Set additional project information
set(COMPANY "JCP353")
set(COPYRIGHT "Copyright (c) 2022 Joel Pepper")
set(IDENTIFIER "com.pepper.Vtkdemo")

set(SOURCE_FILES
    src/main.cpp
    src/mainwindow.cpp
)

add_project_meta(META_FILES_TO_INCLUDE)

# Find libraries
find_package(Qt5 COMPONENTS Widgets REQUIRED)
find_package(VTK REQUIRED)

include_directories(${CMAKE_CURRENT_BINARY_DIR} ${VTK_INCLUDE_DIRS} ${JSON_INCLUDE_DIR})

add_executable(${PROJECT_NAME} ${OS_BUNDLE} ${SOURCE_FILES} ${META_FILES_TO_INCLUDE} ${RESOURCE_FILES}
)

target_precompile_headers(${PROJECT_NAME} INTERFACE QtWidgets.h)

target_link_libraries(${PROJECT_NAME} Qt5::Widgets ${VTK_LIBRARIES})
```

DREXEL UNIVERSITY
College of
Computing &
Informatics

Initial Setup

- Now I'm going to start showing code for basic VTK tasks, most of it from the demo I'll show later
- A handful of VTK objects need to be initialized once then are reused in most operations: always a renderer and a render window, and for the demo I made we also need a file exporter and render window interactor

Initial Setup

- mainwindow.h:

```
...
private:
    vtkSmartPointer<vtkRenderer> renderer;
    vtkSmartPointer<vtkGenericOpenGLRenderWindow> renderWindow;
    vtkSmartPointer<vtkOBJExporter> exporter;
    vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor;
...
```

- mainwindow.cpp:

```
...
renderer = vtkSmartPointer<vtkRenderer>::New();
vtkNew<vtkNamedColors> colors;
renderer->SetBackground(colors->GetColor3d("Black").GetData());
renderWindow = vtkSmartPointer<vtkGenericOpenGLRenderWindow>::New();
renderWindow->AddRenderer(renderer);
ui->qvtkWidget->setRenderWindow(renderWindow);
renderWindowInteractor = renderWindow->GetInteractor();
vtkNew<vtkCallbackCommand> clickCallback;
clickCallback->SetCallback(clickCallbackFunction);
clickCallback->SetClientData(this);
renderWindowInteractor->AddObserver(vtkCommand::LeftButtonPressEvent,
            clickCallback);
...
```

- **vtkRenderer**: Contains actor objects and is associated with the *vtkRenderWindow*

UNIVERSITY of puting & rmatics

Initial Setup

- mainwindow.h:

```
...
private:
    vtkSmartPointer<vtkRenderer> renderer;
    vtkSmartPointer<vtkGenericOpenGLRenderWindow> renderWindow;
    vtkSmartPointer<vtkOBJExporter> exporter;
    vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor;
...
```

- mainwindow.cpp:

```
...
renderer = vtkSmartPointer<vtkRenderer>::New();
vtkNew<vtkNamedColors> colors;

renderer->SetBackground(colors->GetColor3d("Black").GetData());

renderWindow = vtkSmartPointer<vtkGenericOpenGLRenderWindow>::New();
renderWindow->AddRenderer(renderer);
ui->qvtkWidget->setRenderWindow(renderWindow);

renderWindowInteractor = renderWindow->GetInteractor();
vtkNew<vtkCallbackCommand> clickCallback;
clickCallback->SetCallback(clickCallbackFunction);
clickCallback->SetClientData(this);
renderWindowInteractor->AddObserver(vtkCommand::LeftButtonPressEvent,
            clickCallback);
...
```

- **vtkRenderWindow**: The object associated with the Qt window that actually renders our scene

UNIVERSITY of puting & rmatics

Initial Setup

- mainwindow.h:

```
...
private:
    vtkSmartPointer<vtkRenderer> renderer;
    vtkSmartPointer<vtkGenericOpenGLRenderWindow> renderWindow;
    vtkSmartPointer<vtkOBJExporter> exporter;
    vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor;
...
```

- mainwindow.cpp:

```
...
exporter = vtkSmartPointer<vtkOBJExporter>::New();
exporter->SetActiveRenderer(renderer);
exporter->SetRenderWindow(renderWindow);
const char *objFileName = "fish_scene";
exporter->SetFilePrefix(objFileName);
...
```

- **vtkOBJExporter**: Used to export a scene as an .obj file

DREXEL UNIVERSITY
College of
Computing &
Informatics

Initial Setup

- mainwindow.h:

```
...
private:
    vtkSmartPointer<vtkRenderer> renderer;
    vtkSmartPointer<vtkGenericOpenGLRenderWindow> renderWindow;
    vtkSmartPointer<vtkOBJExporter> exporter;
    vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor;
...
```

- mainwindow.cpp:

```
...
renderer = vtkSmartPointer<vtkRenderer>::New();
vtkNew<vtkNamedColors> colors;

renderer->SetBackground(colors->GetColor3d("Black").GetData());

renderWindow = vtkSmartPointer<vtkGenericOpenGLRenderWindow>::New();
renderWindow->AddRenderer(renderer);
ui->qvtkWidget->setRenderWindow(renderWindow);

renderWindowInteractor = renderWindow->GetInteractor();
vtkNew<vtkCallbackCommand> clickCallback;
clickCallback->SetCallback(clickCallbackFunction);
clickCallback->SetClientData(this);
renderWindowInteractor->AddObserver(vtkCommand::LeftButtonPressEvent,
            clickCallback);
...
```

- **vtkRenderWindowInteractor**: Enables user click (and other types) of events within the VTK window

VTK Features Covered in this Talk

I will be covering:

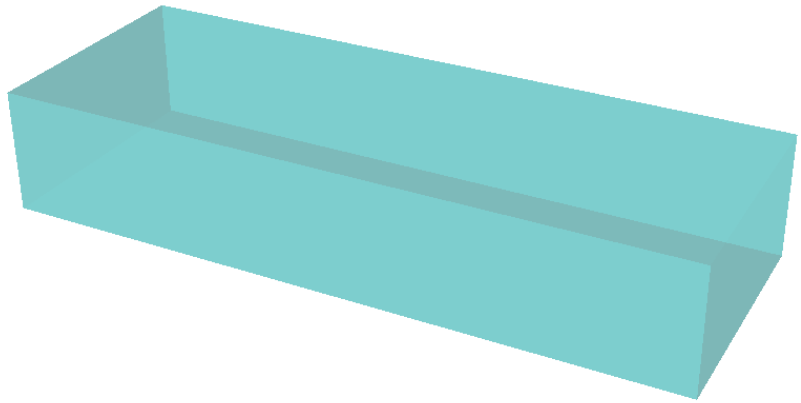
- 4 ways of creating Actors:
  - Basic primitives (e.g. cube, sphere, cylinder)
  - Curves
  - Triangle Meshes from vertex and face matrices
  - Reading triangle Meshes from an `.obj` file
- Exporting VTK scenes to an `.obj` file
- Handling VTK events
- "Picking" of actors via mouse clicks

DREXEL UNIVERSITY
College of
Computing &
Informatics

Cube

`mainwindow.cpp`:

```
vtkNew<vtkCubeSource> cubeSource;
cubeSource->SetBounds(-20.0, 20.0, -10.0, 10.0, -10.0, 100.0);
vtkNew<vtkPolyDataMapper> mapper;
mapper->SetInputConnection(cubeSource->GetOutputPort());
vtkNew<vtkActor> cube;
cube->SetMapper(mapper);
cube->GetProperty()->SetColor(
        colors->GetColor3d("Aqua").GetData());
cube->GetProperty()->SetOpacity(0.3);
renderer->AddActor(cube);
```



DREXEL UNIVERSITY College of Computing & Informatics

Sphere

Not from the demo:

```
vtkNew<vtkSphereSource> sphereSource;
sphereSource->SetCenter(1.0, 2.0, 3.0);
sphereSource->SetRadius(10.0);
sphereSource->SetPhiResolution(10);
sphereSource->SetThetaResolution(10);
vtkNew<vtkPolyDataMapper> mapper;
mapper->SetInputConnection(sphereSource->GetOutputPort());
vtkNew<vtkActor> actor;
actor->SetMapper(mapper);
actor->GetProperty()->SetColor(
        colors->GetColor3d("Red").GetData());
renderer->AddActor(actor);
```

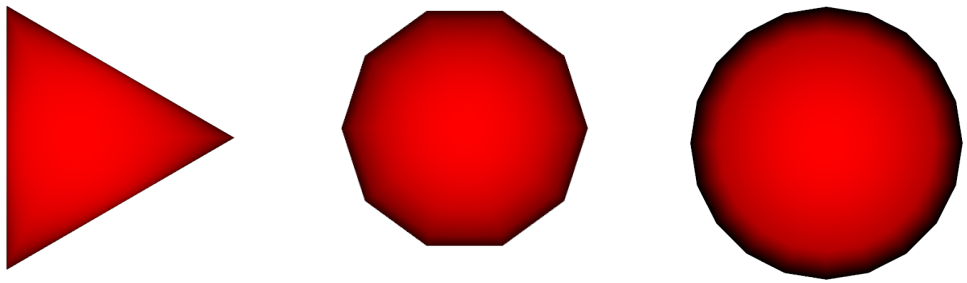
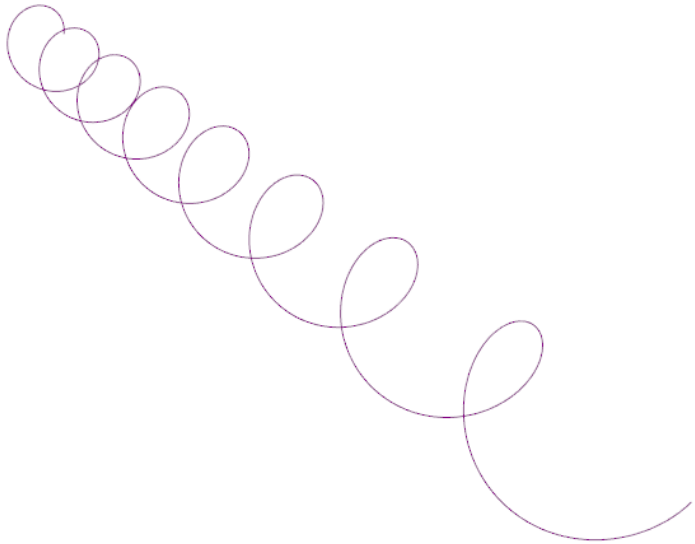


Figure: From left to right: $\phi = 20$ and $\theta = 1$, $\phi = 10$ and $\theta = 10$, $\phi = 5$ and $\theta = 20$

Curve

mainwindow.cpp:

```
    vtkNew<vtkPoints> points;
    for (auto point : fishJS["path"])
        points->InsertNextPoint(point[0], point[1], point[2]);
    vtkNew<vtkPolyLine> polyLine;
    int ptCount = fishJS["path"].size();
    polyLine->GetPointIds()->SetNumberOfIds(ptCount);
    for (int i = 0; i < ptCount; i++)
        polyLine->GetPointIds()->SetId(i, i);
    vtkNew<vtkCellArray> cells;
    cells->InsertNextCell(polyLine);

    // Create a polydata to store everything in
    vtkNew<vtkPolyData> polyData;

    // Add the points to the dataset
    polyData->SetPoints(points);

    // Add the lines to the dataset
    polyData->SetLines(cells);

    // Setup actor and mapper
    vtkNew<vtkPolyDataMapper> mapperLine;
    mapperLine->SetInputData(polyData);

    vtkNew<vtkActor> actorLine;
    actorLine->SetMapper(mapperLine);
    actorLine->GetProperty()->SetColor(
        colors->GetColor3d("Purple").GetData());
    renderer->AddActor(actorLine);
```



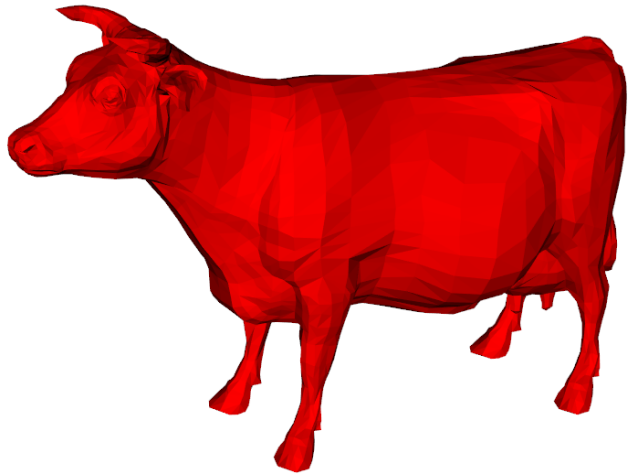
The highlighted line is a JSON object

DREXEL UNIVERSITY College of Computing & Informatics

Triangle Mesh from Matrices

Not from the demo:

```
vtkNew<vtkNamedColors> colors;
MatrixXd verts = myModel->getVertices();
MatrixXi faces = myModel->getFaces();
faces = faces - MatrixXi::Ones(faces.rows(), faces.cols());
vtkNew<vtkPoints> points;
for (auto pt : verts.rowwise()) {
    points->InsertNextPoint(pt(0), pt(1), pt(2));
}
vtkNew<vtkCellArray> triangles;
for (auto t : faces.rowwise()) {
    vtkNew<vtkTriangle> triangle;
    triangle->GetPointIds()->SetId(0, t(0));
    triangle->GetPointIds()->SetId(1, t(1));
    triangle->GetPointIds()->SetId(2, t(2));
    triangles->InsertNextCell(triangle);
}
// Create a polydata object
vtkNew<vtkPolyData> polyData;
// Add the geometry and topology to the polydata
polyData->SetPoints(points);
polyData->SetPolys(triangles);
vtkNew<vtkPolyDataMapper> mapper;
mapper->SetInputData(polyData);
vtkNew<vtkActor> actor;
actor->SetMapper(mapper);
actor->GetProperty()->SetColor(
        colors->GetColor3d("Red").GetData());
renderer->AddActor(actor);
```



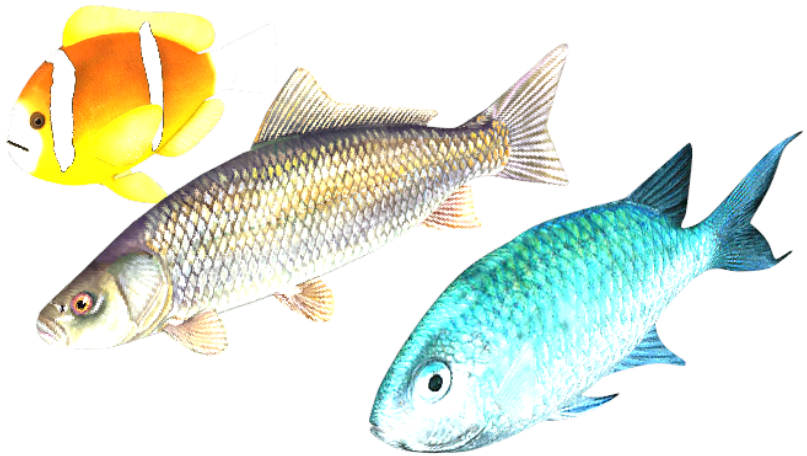
The highlighted lines are from the matrix library Eigen (not otherwise covered in this talk)

DREXEL UNIVERSITY College of Computing & Informatics

Importing CAD Files

mainwindow.cpp:

```
for (auto fish : js) {
    vtkNew<vtkOBJImporter> importer;
    std::string fileName = "../res/" + fish["file"].get<std::string>() + ".obj";
    // NOTE: this assumes you are running the program from codefest22-vtk-demo/build
    importer->SetFileName(fileName.c_str());
    importer->SetRenderWindow(renderWindow);
    importer->Update();
}
```

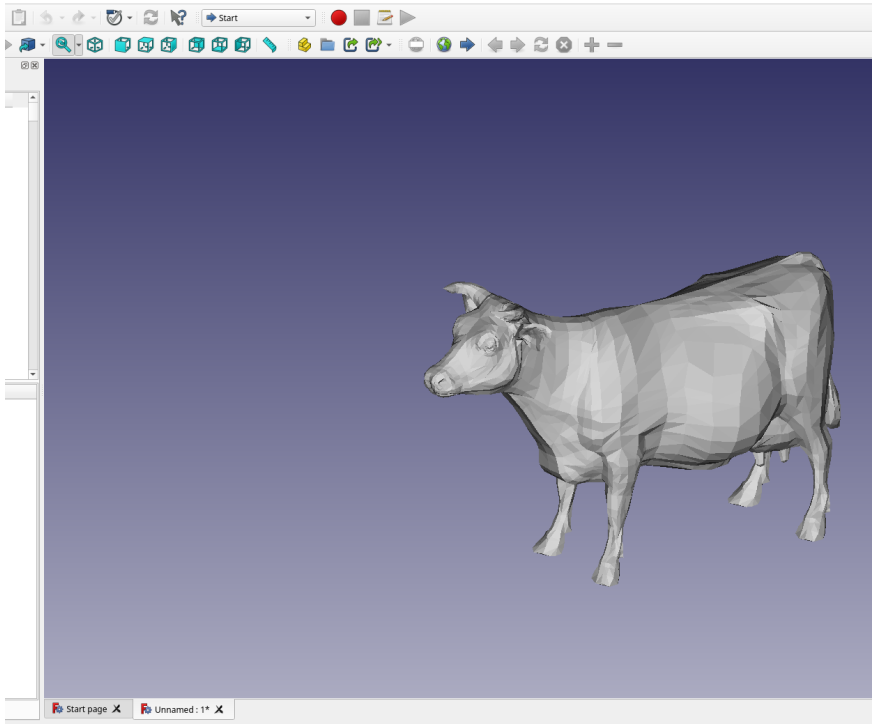


DREXEL UNIVERSITY College of Computing & Informatics

Exporting CAD Files

mainwindow.cpp:

```
exporter = vtkSmartPointer<vtkOBJExporter>::New();
exporter->SetActiveRenderer(renderer);
exporter->SetRenderWindow(renderWindow);
const char *objFileName = "fish_scene";
exporter->SetFilePrefix(objFileName);
exporter->Update();
```



DREXEL UNIVERSITY College of Computing & Informatics

Click Event Handler

mainwindow.cpp:

```
void clickCallbackFunction(vtkObject* caller,
        long unsigned int eventId,
        void* clientData,
        void* vtkNotUsed(callData))
{
    vtkNew<vtkNamedColors> colors;
    auto mainWindow = reinterpret_cast<MainWindow*>(clientData);
    auto interactor = reinterpret_cast<vtkRenderWindowInteractor*>(caller);
    int* clickPos = interactor->GetEventPosition();
    cout << "Click callback" << endl;
    cout << "Event: " << vtkCommand::GetStringFromEventId(eventId)
        << "\nClick Position: " << clickPos[0] << ", " << clickPos[1]
        << std::endl;
    // Pick from this location.
    vtkNew<vtkPropPicker> picker;
    auto renderer = interactor->GetRenderWindow()->GetRenderers()->GetFirstRenderer();
    auto cube = renderer->GetActors()->GetLastActor();
    vtkNew<vtkActor> cubeCopy;
    cubeCopy->ShallowCopy(cube);
    renderer->RemoveActor(cube);
    picker->Pick(clickPos[0], clickPos[1], 0, renderer);
    renderer->AddActor(cubeCopy);

    double* pos = picker->GetPickPosition();
    cout << "Pick Position: " << pos[0] << " "
        << pos[1] << " " << pos[2] << std::endl;

    auto pickedActor = picker->GetActor();
    if (pickedActor == nullptr) {
        cout << "No actor picked." << std::endl;
    } else {
        cout << "Picked actor: " << picker->GetActor() << std::endl;
        mainWindow->getInfoBox()->setPlainText(
            QString::fromStdString(mainWindow->getFishInfo(pickedActor)));
    }
}
```

DREXEL UNIVERSITY College of Computing & Informatics

Fish Movement C++ Demo

- The C++ demo I have consists of visualizing data on "fish swimming patterns" (that I made up and generated myself)
- 3 fish are defined in a `.json` file with their models, some transformations on those models, info about the species and the path they swim
- The visualization aspect includes rendering a translucent box to represent a river, loading the fish models, fitting curves to the swim paths, then animating the fish models along those paths
- Picking of fish is also included to show their information in the Qt window, as well as a few basic operations bound to Qt buttons

Philly Building Topography Python Demo

- This demo visualizes building height data I got from [3]
- I took ~ 2000 buildings and sent them through an address to latitude longitude converter, then graphed the buildings as rectangles in VTK
- There's also a (very) rough outline of Philadelphia, and the buildings can be clicked on to show their information in a Qt text box

3D Network Graph JavaScript Demo

- In this demo I show how to dynamically generate a 3D network graph using `VTK.js`
- Buttons can be used to add and remove nodes from the network, and connections are also added and severed appropriately

(Finally) The End!

- There's of course a ton more VTK can do but this should give you a good starting point for making use of it
- Now, hopefully there's time for questions

DREXEL UNIVERSITY
College of
Computing &
Informatics


"Using VTK to Visualize Scientific Data (online tutorial)," https://www.bu.edu/tech/support/research/training-consulting/online-tutorials/vtk/#VISPIPE.


"Minimal CMake Template for Qt 5 Projects," https://github.com/euler0/mini-cmake-qt/tree/qt5.


"Building Footprints," https://www.opendataphilly.org/dataset/buildings.