ECE 462
Object-Oriented Programming using C++ and Java

Lecture 21

Yung-Hsiang Lu
yunglu@purdue.edu

Testing Strategy

- Testing is one, not the only one, step to ensure quality.
- Before writing code, think about how to test it.
- Do not be surprised that you write more code for testing than for the project.
- Danger of using testing to ensure quality: you usually test what you suspect. The program usually breaks at places where you are confident.
- Sometimes, reading code line-by-line can find and fix problems faster than writing testing code, especially for multi-thread programs.

Design and Testing

code	unit test
design	integration test
requirements	validation test
system engineering	system testing

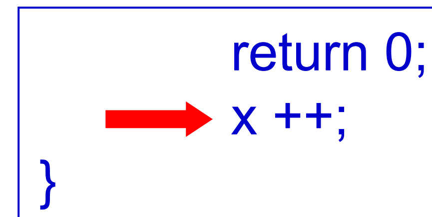
familiar to most of you (unit test)

often the hardest (system testing)

- Importance of unit testing: well-designed software should allow only **limited visibility** (encapsulation) for better consistency. Hence, testing from outside is difficult.
- Build software in **layers**. A lower layer should be fully tested before building a higher layer.

Test Coverage

- How much code is exercised in a test? How many possible paths are traversed?
- Many tools exist for reporting code coverage.
- Low coverage: not fully tested ⇒ bad test
- High coverage: can **hardly** test each possible **path** ⇒ quality **unclear**
- Testing discover many problems? good or bad?
- "dead code": code that is impossible to reach, usually indicates design or coding errors



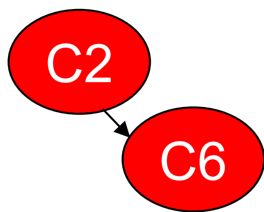
Execution Path

C1
C2
C3
C4
C5
C6
C7

C1
C2
C3
C4
C5
C6
C7

C1
C2
C3
C4
C5
C6
C7

C: code fragment



Every node has been visited (100% test coverage) but C2→C6 has not been tested

Quantify Testing Quality

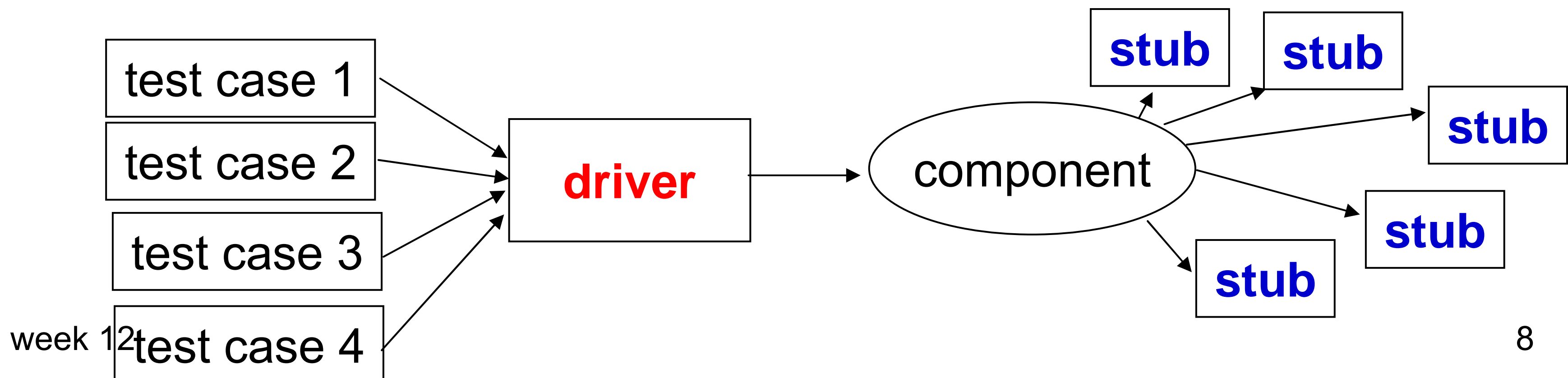
- coverage: (% code and % paths) of the test
- efficiency: evenly distribute? time to cover 99%?
- progression: % new code tested
- discovery rate: % bugs found for every line of test code
- configurability: selections of features to test
- ratio: how much testing code needed to test actual code
- expandability: amount of efforts needed to test new features
- degree of automation: can it be fully automated, semi-automated, or complete manual?

Testing Steps

- unit testing, integration testing, regression
- unit testing:
  - individual components
  - often conducted by the developer
  - often using dedicated testing code to create input data, exercise the components
  - often traced by single steps
  - check boundary conditions and error handling
  - check interface correctness and responses to incorrect inputs

Unit Testing

– examine the performance
– should be performed before "cvs commit"
– should be put into the repository
– should be configurable for related components
– require careful planning **in advance**
– driver: code to call the component, stub: code to be called by the component. both are overhead

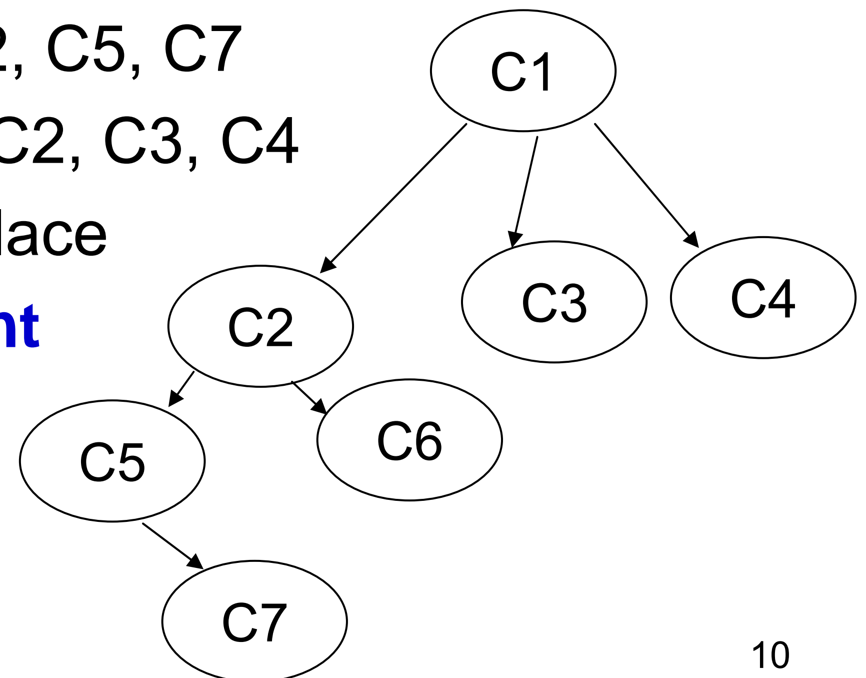


Integration Testing

- Interface incompatibility is often the reason software breaks. incompatibility ≠ compiler error
- types of interface errors, even passing compilation, e.g.
  - wrong types (object of derived class or base class)
  - wrong assumptions, for example
    - who is responsible for allocating or releasing memory
    - who may modify the data, especially global variables
    - sorted or nearly sorted? wrong result or wasting time
  - wrong timing assumption for real-time software
- **incremental** integration: add one component (e.g. class) each time
- may still use drivers and stubs (how do you know they are correct?)

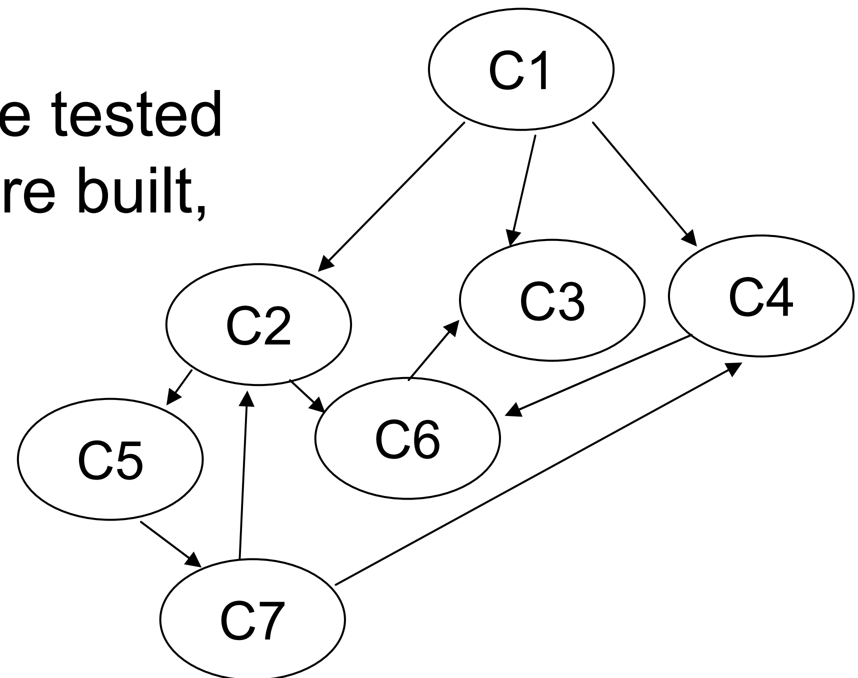
Top-Down Integration

- using control flow to determine the integration order
- starting from the main component ("main" in C/C++) as the driver
- integrate callees (replace stubs) of the main component
- depth-first integration: C1, C2, C5, C7
- breadth-first integration: C1, C2, C3, C4
- after one successful test, replace a **stub** by the real **component**
- regression test (later) to ensure tested components still work



Challenges in Top-Down Testing

- **control flow** and **call graph** are not downward only or acyclic
- many functionalities cannot be tested before the leaf components are built, e.g. C1 needs the data (or objects) generated in C7 to test C3
- depth-first or breath-first only may not represent normal execution paths



Bottom-Up Integration

1. start from individual components (such as classes)
2. put several components together, use a driver to test them
3. replace the driver by a real component
4. repeat step 2-3

- difficulties:
  - which components to integrate first? They must have a common driver.
  - control may not be upward only or acyclic
  - required data (or objects) may be generated from a component that has not been integrated

Regression Testing

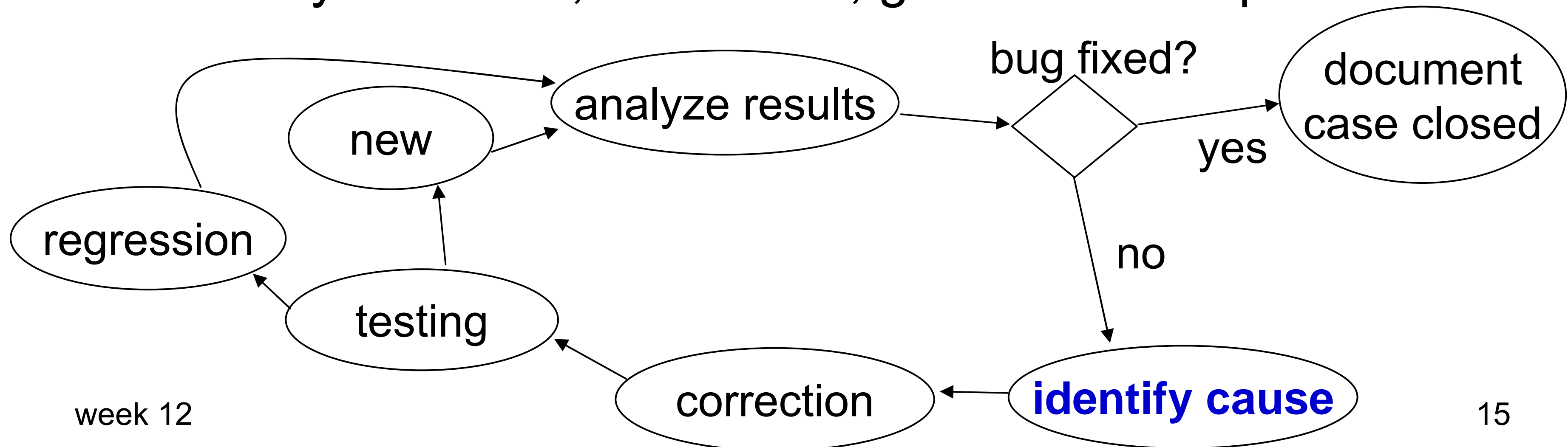
- re-test what has been tested after new components are integrated into the project
- (same) test after errors are corrected
- **expandable** as more components are added
- **configurable** so that new components can be exercised more
- should be automated as much as possible (consider using cron jobs)
- Most important / frequently used features should be test more thoroughly.

Test Documentation

- Testing should be planned and documented.
  - test plan: what to test, when to test, who runs the test, how to run the test, what data to use …
  - testing with integration: how components are integrated, regression testing procedure
  - procedure to test: manual, automated, or semi-automated? conditions, tools, special hardware …
  - test result analysis: what to expect, how to diagnose
  - test result management: providing a trace of integration and testing
  - re-test procedure after correction

Hypothesis-Test Debugging

- Most software developers take "hypothesis-testing" approach for debugging:
  - guess what is the cause ⟵ **usually the hardest part**
  - modify the code
  - run some tests
  - analyze results, if not fixed, guess another place



Time-Sensitive or Massive Data

- Single-step code is not always the best way to debug.
- Some programs cannot be single-stepped:
  - time-sensitive, interacting with the physical world. It does not wait for the program.
  - massive amount of data, too many steps. How do you single-step an image with 3 million pixels?
- Detect error conditions before proceeding. **Always** check the return value of a system or library call, such as connect, read, write, malloc, fork …
- Create increasingly complex and realistic testing data: smaller images, simpler images, single color, checkerboard …

Time-Sensitive or Massive Data

- Detect and handle errors before they propagate.
- Generate execution logs for post-execution analysis. Be careful about the impact on timing.
- controversy of "assert": assert (something must be true);

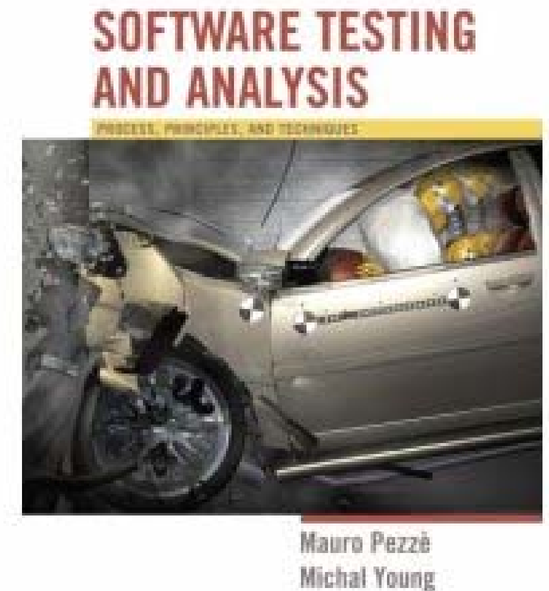
```
assert (x > 0);
```

Program stops immediately if the condition fails

$\Rightarrow$ errors do not propagate.

$\Rightarrow$ users cannot recover anything, especially lost data.

Problem in Testing OOP



ISBM 978-0-471-4593-6

- main challenge in most tests: very large space for possible execution path
- Each "if" represents a branch of execution. Can you test all possible paths of execution?
- implicit branches from polymorphism

```
BaseClass obj = new BaseClass;
obj.method(); // BaseClass'
obj = new DerivedClass;
obj.method(); // DerivedClass'
```

- state-dependent behavior
- inheritance
- abstract class
- polymorphism
- exception handling
- concurrency
- encapsulation

Testing OOP

- develop use cases to ensure the intended functionalities are correct
- create test cases based on the sequence diagrams in the design
- traverse all states an object may have
- identify the possible messages between classes
- intraclass testing (unit test)
  - each class (except abstract) must be instantiated (i.e. create an object)
  - invoke polymorphic calls
  - explicitly throw exceptions to trigger the handling code
- interclass testing
  - test class hierarchies incrementally
  - use layered approach

Layered Structure

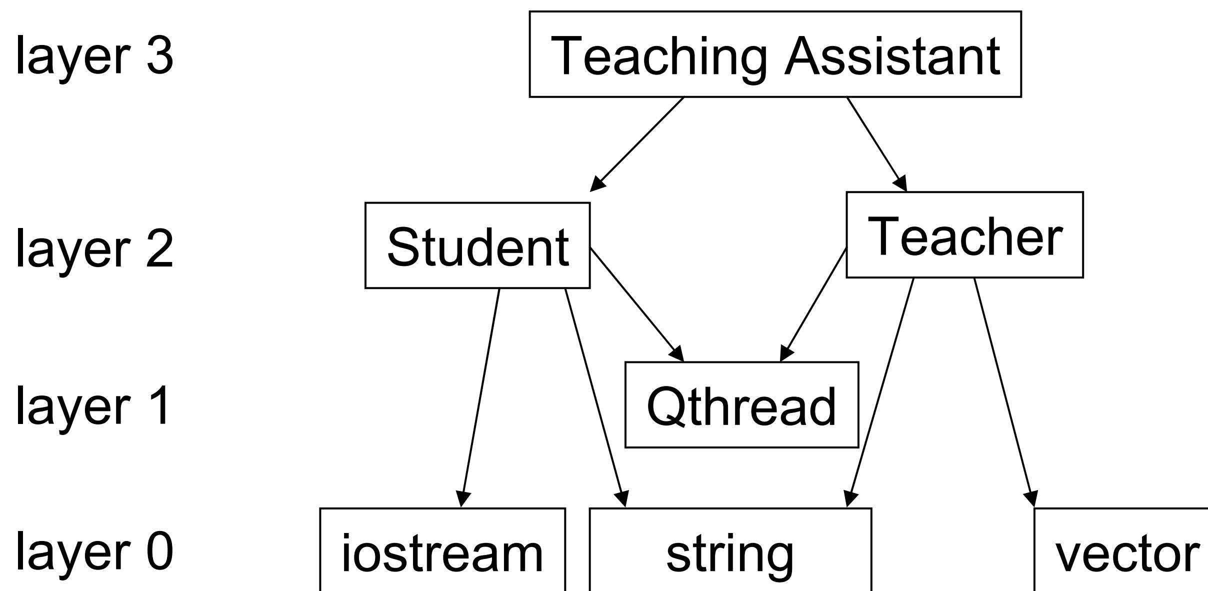
- Structure the program so that each file can be assigned a unique layer number.
- Layer 0: files from language, such as iostream
- Layer 1: library files, such as Qt's classes

distinction not precise

- Layer 2: common files used in multiple projects in your organization
- Layer 3: stable files used for months
- Layer 4: recently developed and test files
- Layer 5: unstable files
- Layer n: depends on files in layer 0, 1, 2, …, n-1

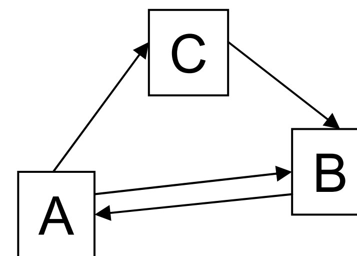
File Layers

A file is assigned layer n if it depends on only files in layer 0, 1, 2, …, n-1



Why to Layer Files / Classes

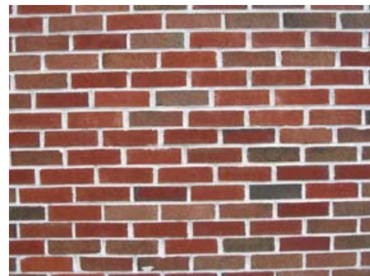
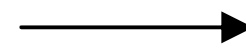
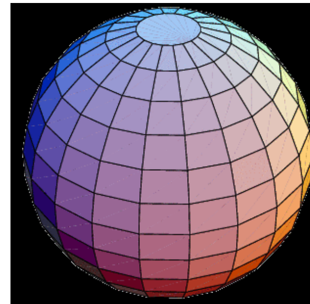
- A file with a lower layer number should be more stable and have a higher degree of correctness
- Strictly layered structure allow **unit testing** a recently developed module in the program
- Layering indicates the **precedence** of development. If a module is a foundation for some other modules, this module should be placed (physically) in a file that has a lower layer number.
- Cyclic dependence often suggests flaws in **logical** design.



Texture Mapping
(wrap an image to a curve surface)

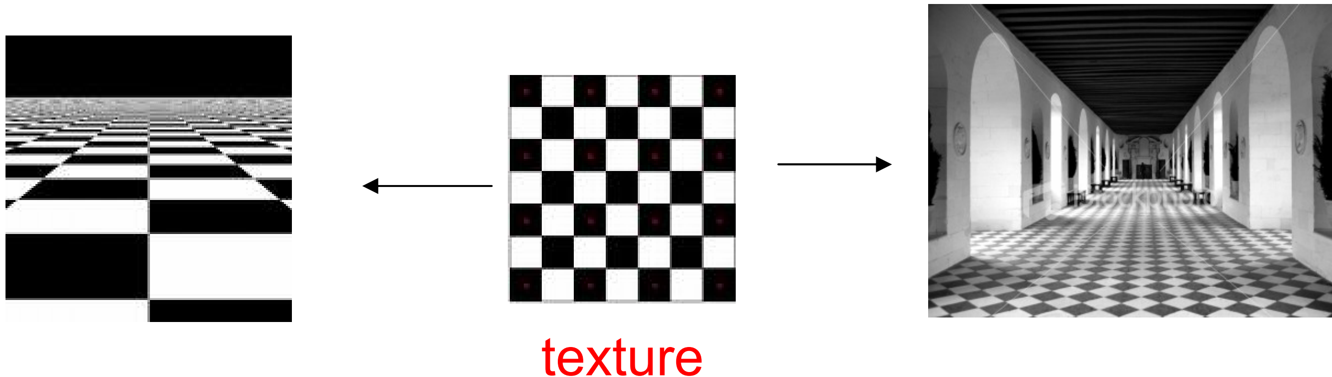
Texture Mapping

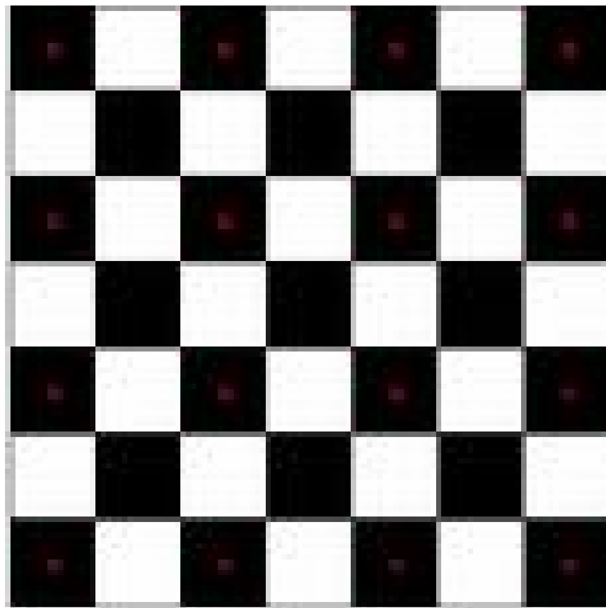
give the impression of the surface properties, usually on a non-rectangular and 3D surface

Rendering Surface

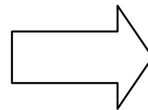
- Theoretically, it is possible to render the surface, for example of grass, by
  1. modeling each object using triangles
  2. calculating the lighting and shading
  3. eliminate occluded pixels based on depth
- In reality, this approach takes too long.
- procedure: from a pixel's location, look up (map) the pixel from the texture

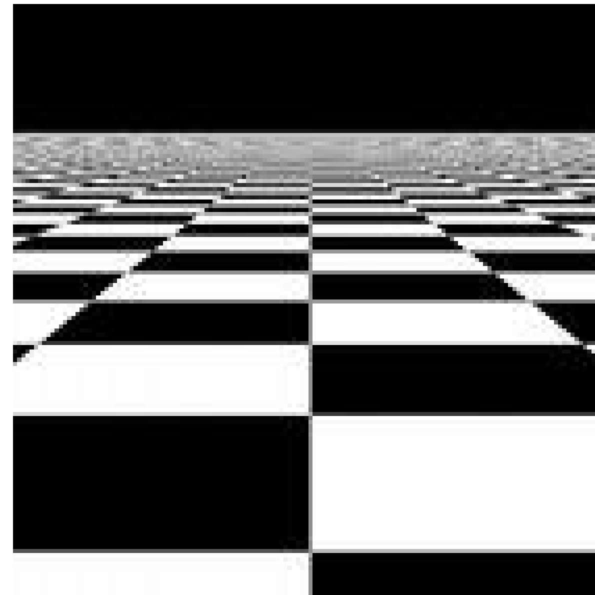




Pattern Image

I(u,v)





Target

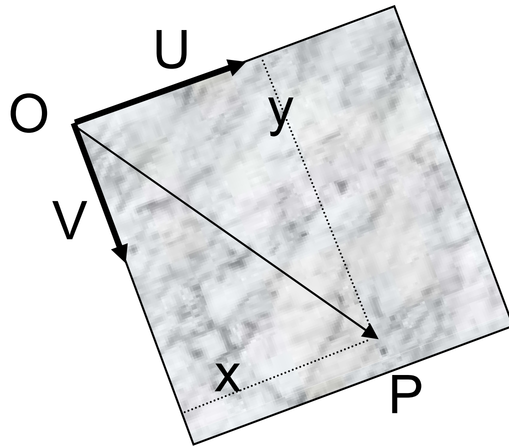
P(x,y)

- Find a function f such that f: texel (u,v) $\rightarrow$ pixel (x,y).
- It is usually more efficient to find g: (x,y) $\rightarrow$ (u,v). What to do if u or v is not an integer?

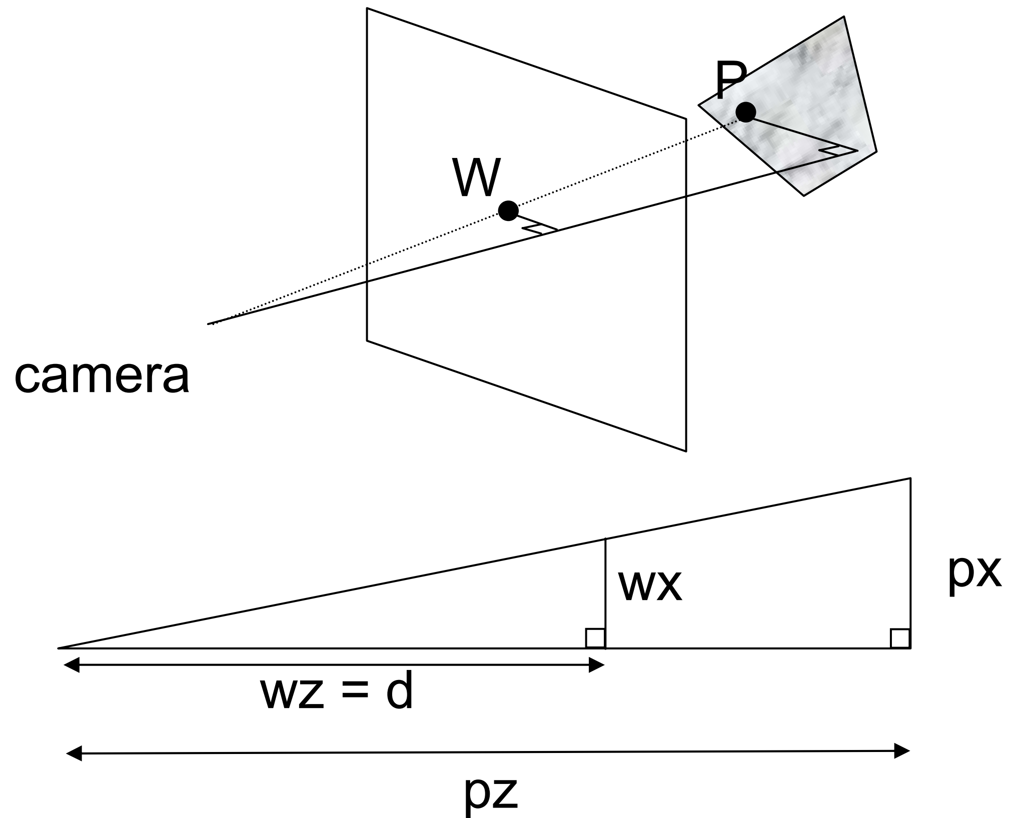
Foundation of Texturing

$$x = \vec{U} \cdot \overrightarrow{PO}$$

$$y = \vec{V} \cdot \overrightarrow{PO}$$



target
(pixel)



$$\vec{W} = \vec{P}\frac{d}{pz}$$

Let N be the normal vector for the surface.

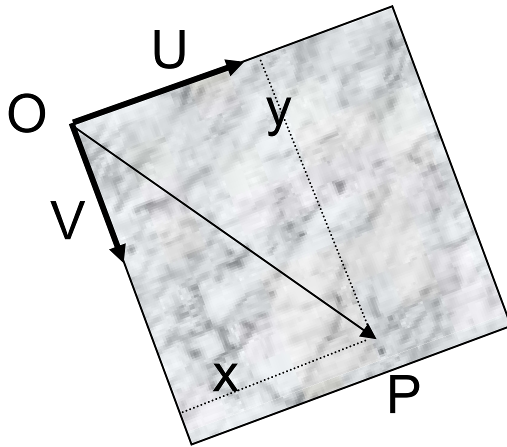
$$\vec{N} \cdot \overrightarrow{PO} = 0$$

$$\vec{N} \cdot (\vec{O} - \vec{P}) = 0$$

$$\vec{N} \cdot \vec{O} = \vec{N} \cdot \vec{P}$$

$$\vec{W} = \vec{P}\frac{d}{pz} \qquad \vec{P} = \vec{W}\frac{pz}{d}$$

$$\vec{N} \cdot \vec{O} = \vec{N} \cdot \vec{W}\frac{pz}{d} \qquad \frac{pz}{d} = \frac{\vec{N} \cdot \vec{O}}{\vec{N} \cdot \vec{W}}$$



$$\vec{P} = \vec{W}\frac{\vec{N} \cdot \vec{O}}{\vec{N} \cdot \vec{W}}$$

$$x = \vec{U} \cdot (\vec{W}\frac{\vec{N} \cdot \vec{O}}{\vec{N} \cdot \vec{W}} - \vec{O})$$

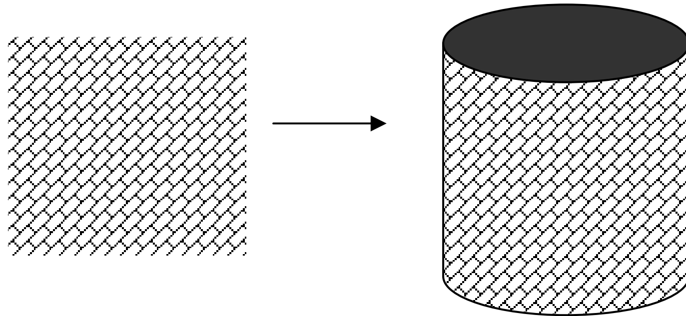
$$y = \vec{V} \cdot (\vec{W}\frac{\vec{N} \cdot \vec{O}}{\vec{N} \cdot \vec{W}} - \vec{O})$$

Simple Examples

- linear texturing

$$x = au + bv + c$$
$$y = du + ev + f$$

- square to cylinder



$$x = r\cos(2\pi u)$$
$$y = r\sin(2\pi u)\cos(2\pi v)$$
$$z = r\sin(2\pi u)\sin(2\pi v)$$

Steps for Texturing in Java 3D

- prepare texture images
  - Java3D requires that an image size to be a power of 2 (2, 4, 8, 16 ...) for efficiency
  - accepted format: JPEG and GIF
- load the texture
- set the texture in Appearance bundle, create a geometric shape as the target for texture
- specify TextureCoordinate of Geometry

```
/** * SimpleTextureSpinApp creates a single plane with texture mapping. */
public class SimpleTextureSpinApp extends Applet {
    BranchGroup createScene() {
      BranchGroup objRoot = new BranchGroup();
      Transform3D transform = new Transform3D();
      QuadArray plane = new QuadArray(4, GeometryArray.COORDINATES
                              | GeometryArray.TEXTURE_COORDINATE_2);
      Point3f p = new Point3f(-1.0f,  1.0f,  0.0f);
      plane.setCoordinate(0, p);
      p.set(-1.0f, -1.0f,  0.0f);
      plane.setCoordinate(1, p);
      p.set(1.0f, -1.0f,  0.0f);
      plane.setCoordinate(2, p);
      p.set(1.0f,  1.0f,  0.0f);
      plane.setCoordinate(3, p);
      TexCoord2f q = new TexCoord2f( 0.0f,  1.0f);
      plane.setTextureCoordinate(0, 0, q);
```

```
q.set(0.0f, 0.0f);
plane.setTextureCoordinate(0, 1, q);
q.set(1.0f, 0.0f);
plane.setTextureCoordinate(0, 2, q);
q.set(1.0f, 1.0f);
plane.setTextureCoordinate(0, 3, q);
Appearance appear = new Appearance();
String filename = "stripe.gif";
TextureLoader loader = new TextureLoader(filename, this);
ImageComponent2D image = loader.getImage();
if(image == null) {
    System.out.println("load failed for texture: "+filename);
}
// can't use parameterless constuctor
Texture2D texture = new Texture2D(Texture.BASE_LEVEL,
      Texture.RGBA,   image.getWidth(),  image.getHeight());
texture.setImage(0, image);
appear.setTexture(texture);
```

```
appear.setTransparencyAttributes(
    new TransparencyAttributes(TransparencyAttributes.FASTEST,  0.1f));
Shape3D planeObj = new Shape3D(plane, appear);
// rotate object has composited transformation matrix
Transform3D rotate = new Transform3D();
Transform3D tempRotate = new Transform3D();
rotate.rotX(Math.PI/4.0d);
tempRotate.rotY(Math.PI/5.0d);
rotate.mul(tempRotate);
TransformGroup objRotate = new TransformGroup(rotate);
// Create the transform group node and initialize it to the
// identity.  Enable the TRANSFORM_WRITE capability so that
// our behavior code can modify it at runtime.  Add it to the
// root of the subgraph.
TransformGroup objSpin = new TransformGroup();
objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
objRoot.addChild(objRotate);
objRotate.addChild(objSpin);
```

```
// Create a simple shape leaf node, add it to the scene graph.
// ColorCube is a Convenience Utility class
objSpin.addChild(planeObj);
Transform3D yAxis = new Transform3D();
Alpha rotationAlpha = new Alpha(-1, 4000);
RotationInterpolator rotator =
  new RotationInterpolator(rotationAlpha, objSpin, yAxis,
                                   0.0f, (float) Math.PI*2.0f);
// a bounding sphere specifies a region a behavior is active
// create a sphere centered at the origin with radius of 1
BoundingSphere bounds = new BoundingSphere();
rotator.setSchedulingBounds(bounds);
objSpin.addChild(rotator);
Background background = new Background();
background.setColor(1.0f, 1.0f, 1.0f);
background.setApplicationBounds(new BoundingSphere());
objRoot.addChild(background);
```

```
        return objRoot;
    }
    public SimpleTextureSpinApp (){
      setLayout(new BorderLayout());
      GraphicsConfiguration config =  SimpleUniverse.getPreferredConfiguration();
      Canvas3D canvas3D = new Canvas3D(config);
      add("Center", canvas3D);
      canvas3D.setStereoEnable(false);
      SimpleUniverse u = new SimpleUniverse(canvas3D);
      // This will move the ViewPlatform back a bit so the
      // objects in the scene can be viewed.
      u.getViewingPlatform().setNominalViewingTransform();
      u.addBranchGraph(createScene());
    }
    public static void main(String argv[])
    {
      System.out.print("SimpleTextureSpinApp.java \n- ");
```

```
    System.out.println("The simpliest example of using texture mapping.\n");
    System.out.println("This is a simple example progam from The Java 3D API
     Tutorial.");
    System.out.println("The Java 3D Tutorial is available on the web at:");
    System.out.println("http://java.sun.com/products/java-media/3D/collateral ");
    new MainFrame(new SimpleTextureSpinApp(), 256, 256);
  }
}
```