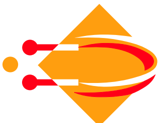
Input Handling, Animation, and Game Objects

Kim Steenstrup Pedersen

kimstp@itu.dk

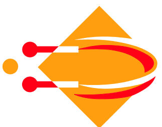
www.itu.dk/courses/SSPG/F2005/

The IT University of Copenhagen

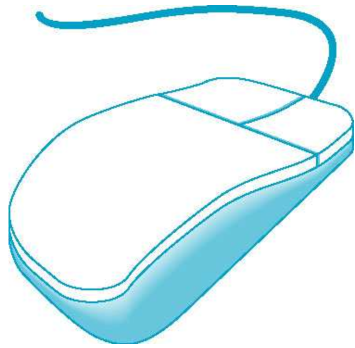


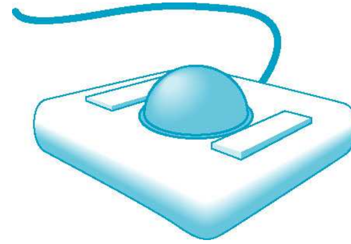
Plan for today

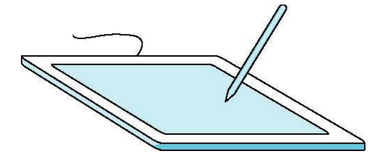
- Input and event handling
- Overview of animation from a computer graphics point of view
- Animation in Panda3d
- Script programming paradigm
- Opening up the game loop (`run()` and Panda tasks)
- Game objects

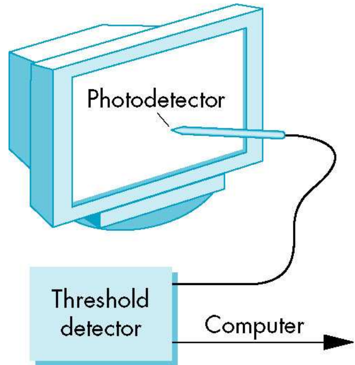


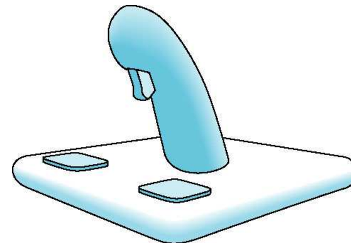
Input and event handling: Some input devices

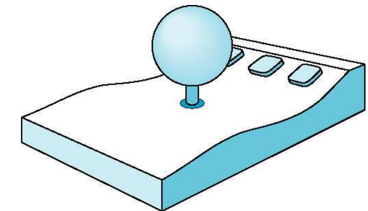


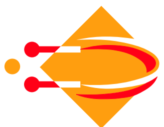












Input and event handling

Input modes in general:

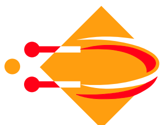
- Request mode
- Sample mode
- Event mode

Event mode:

- Input generates events which are represented as messages / packages. Events are put into an event queue and are processed in a First-In First-Out manner.

Input by callback functions:

- Events are processed through callback functions.
- Your program must register callback functions with an event manager.
- The event manager process the events in the event queue and call the appropriate callback functions.

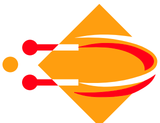


We call a collection of callback functions an event handler.

Input and event handling in Panda3D

In Panda3D event handlers are subclasses of the DirectObject class. The event manager is called `messenger`.

```
# myeventhandler.py
from direct.showbase import DirectObject
class MyEventHandler(DirectObject.DirectObject):
    "MyEventHandler: Handles various events"
    def __init__(self, car):
        self.car = car
        self.accept('arrow_up',self.handleArrowup)
        self.accept('arrow_down',self.handleArrowdown)
        self.accept('arrow_left',self.handleArrowleft)
        self.accept('arrow_right',self.handleArrowright)

    def handleArrowup(self):
        self.car.drive()

    def handleArrowdown(self):
        self.car.reverse()

    def handleArrowleft(self):
        self.car.left()

    def handleArrowright(self):
        self.car.right()
```

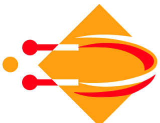


The city and buggy example done right: Object oriented

Lets introduce car objects to store information about a car.

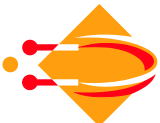
The common car interface:

```
# car.py
# Define the interface for car objects
class Car:
    "Car: Base class for all cars"

    def drive(self):
        pass

    def reverse(self):
        pass

    def left(self):
        pass

    def right(self):
        pass

    def bindCamera(self, camera):
        pass
```



The city and buggy example: Using the event handler

```
# citybuggysimple1.py
from myeventhandler import MyEventHandler

# Set up event handling
evHandler = MyEventHandler(buggy)

# messenger.toggleVerbose() # Debug print all events recieved
                            # by messenger
print messenger # List all registered handlers
messenger.send('mouse1') # Generate and send an event
# messenger.clear() # Clear all handlers registered with
                    # messenger
```



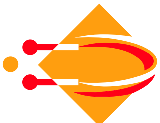
Overview of animation in computer graphics

Definition of animation: Controlled motion of objects and sub-objects.

We would like to produce animations of realistic motion. Realistic motion can be obtained via motion capture equipment, by scripting the motion, or by modelling the motion.

Categorisation of computer animation:

- Rigid body: Keyframing/interpolation, scripted animation, sprites, animating the camera.
- Articulated structures: Kinematics (forward and inverse), skinning.
- Physically based:
  - Rigid body: Using classical mechanics for rigid body motion.
  - Particles: Individual particles are moved under various forces. Good for animation of fire, water, smoke and such.
  - Collision detection and response: Objects should react when they hit other objects.
  - Deformable objects: Animation of water, cloth, flesh etc.
- Behavioural animation: Including behavioural reaction into the animation. Examples: Human and animal motion (Terzopoulos Fish animation), the large battles in Lords of the Rings II – III.



Rigid body animation

Two strategies:

Keyframing / Interpolation: Modelled on the classical approach which involves a chief animator and inbetweeners. Keyframes are specified and motion is interpolated in between.

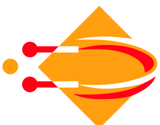
Properties such as position and orientation is interpolated in between keyframes. The keyframes specify e.g. position and orientation at specific time instances.

The problem is to choose interpolation strategy. Furthermore interpolation does not obey the laws of physics.

Scripting: Motion is specified by curves representing the changes of the individual parameters. Example: A curve A for position and a curve B for the change of velocity along the curve A.

The script can be specified in a tool like 3D Studio Max.

The animation curves are usually represented by cubic parametric curves, e.g. B-splines or Bezier curves.



Articulated structure animation

Definition of articulated structure: A collection of rigid bodies which are connected via joints. Example: A stick figure.

Robotics techniques are used for computation of the motion:

Forward kinematics: The motion of the individual rigid bodies is defined by specification of the motion of the individual joints. Example: Specifying the angles between the joints in a stick figure. Scripting could be used to change the parameters over time.

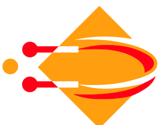
Alternative view: The rigid bodies are transformed relative to each other.

Forward kinematics is usually a tedious and time consuming approach.

Inverse kinematics: Specify the position of the end points of the articulated structure at two time instances A and B. Inverse kinematics can then give a solution for animating the rigid body movement from A to B.

This is a difficult mathematical problem. A solution does not always exist.

The benefit is that the approach is fully automated.



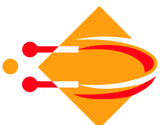
Articulated structure animation: Skinning

An articulated object is represented by a skeleton with a polygon mesh attached to it. The skeleton parts are called the bones and the mesh is called the skin.

The vertices's of the skin mesh is attached to one or more bones.

The bones are moved using e.g. kinematics or scripting techniques and the skin vertices's are moved relative to the bones that they are attached to.

Allow for deformation of the skin mesh at joints.



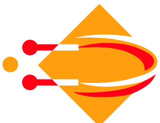
Animation in Panda3d: Actors

Panda supports rigid body animation (both keyframing and scripted animation) and scripted skinning as well as particle animation.

The `Actor` class takes care of skinning animation:

```
from direct.actor import Actor

<NodePath> = Actor.Actor('Model file',
                         {'Animation name':'Animation file'})
# Alternatively
<NodePath> = Actor.Actor()
<NodePath>.loadModel('Model file')
<NodePath>.loadAnim({'Animation name':'animation file'})

# Play the animation
<NodePath>.play('Animation name') # play animation and stop
<NodePath>.loop('Animation name')
<NodePath>.stop('Animation name')
<NodePath>.setPlayRate(<Frame rate>,'Animation name')
```

Actors have other more advanced features, e.g. multiple animations per model and joint manipulation.



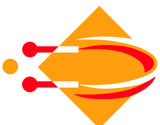
Animation in Panda3d: Intervals

Keyframing / Interpolation is supported via intervals:

```
from direct.interval.IntervalGlobal import *

# Example of linear interpolation interval: Position
<Interval> = <NodePath>.posInterval(<duration>, Point3(x,y,z),
                                    startPos=Point3(x,y,z),...)

# Actor animations can also be converted to an interval
<Interval> = <Actor>.actorInterval('Animation name',
                                   loop = <0 or 1>,
                                   duration=<Duration>, ...)
```

Linear interpolation: $(1-\alpha)A + \alpha B$, $\alpha \in [0,1]$.

All intervals support the methods `start`, `loop`, `finish`.



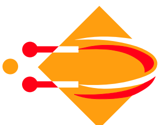
Animation in Panda3d: Sequences and Parallels

Sequences and parallels are intervals that allows for execution of intervals either sequential or in parallel.

```
<sequence> = Sequence(<Interval>, ..., <Interval>,
                      name = 'Sequence name')
<sequence> = Parallel(<Interval>, ..., <Interval>,
                      name = 'Parallel name')
```

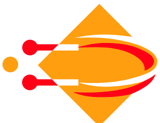
Panda also supports Motion paths for scripted animation, function intervals and more advanced concepts.

Actual animation is done in the C++ core of Panda3d.



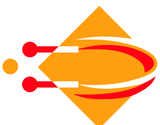
The city and buggy example done right: A buggy

```
# buggy.py
import car
import carfactory
from direct.actor import Actor
from direct.interval.IntervalGlobal import *

class Buggy(car.Car,Actor.Actor):
    def __init__(self):
        self.carMove = Sequence()

        # Initalize the Car and Actor
        car.Car.__init__(self)
        Actor.Actor.__init__(self)

        # Load model
        self.loadModel("buggy/buggy-drive.X")

        # Load animation
        self.loadAnims({"drive":"buggy/buggy-drive.X"})

        # Setup the buggy in the scene graph
        self.setScale(0.05)
        self.setPos(10,-30,0)
        self.setHpr(270,0,0)
```



The city and buggy example done right: A buggy (cont.)

```
def drive(self):
    # Otherwise the animation will start all over again
    if self.carMove.isStopped():
       pos = self.getPos()
       # Rotate v to point in the direction of the car
       # orientation
       R=Mat3.rotateMat(self.getH())
       v = Vec3(0,-1,0) # Orientation of the model
       v = R.xform(v)
       posInterval = self.posInterval(0.5, pos+v*30,
                                      startPos=pos)
       animInterval = self.actorInterval("drive",loop=1,
                                         duration=0.5)
       self.carMove = Parallel(posInterval,
                               animInterval,
                               name = "carMove")
       self.carMove.start()
```

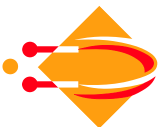


Aside: Panda math engine

Panda supports operations on points, vectors and matrices.

```
from direct.interval.IntervalGlobal import *

R=Mat3.rotateMat(45) # Matrix for 45 degrees rotation around
                     # the z-axis
v = Vec3(0,-1,0)  # Create a 3D vector object
P = Point3(2,1,3) # Create a 3D point object
v = R.xform(v)    # Transform v by R (R v)
P+v               # Adding a point and a vector
```

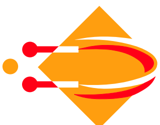
Math in Python: `import math` in order to use functions like `cos` and `sin`.



The city and buggy example done right: A buggy (cont.)

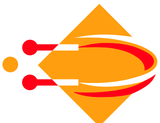
Continue defining the methods of the Buggy class:

```
    def left(self):
        self.setH(self.getH()+5)

    def right(self):
        self.setH(self.getH()-5)

    def bindCamera(self,camera):
        camera.reparentTo(self)
        camera.setPos(0,1475,506)
        camera.setHpr(180,348,0)

# Car factory: Create function and registration
def createBuggy():
    return Buggy()

carfactory.Instance.Register("buggy",createBuggy)
```



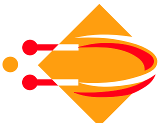
The city and buggy example done right: A car factory

```
# carfactory.py

# Not a singleton class
class CarFactory:
    m_Creators = {}
    def Register(self, sType, aCreator):
        self.m_Creators[sType] = aCreator

    def Create(self, sType):
        f=self.m_Creators[sType]
        return f()


# Pseudo singleton
def Initialize():
    return CarFactory()

Instance = Initialize()
```



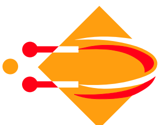
The city and buggy example done right: Driving a car

```
# citybuggy.py
import direct.directbase.DirectStart
from myeventhandler import MyEventHandler
import carfactory
import buggy
import yugo

# Load the city environment model
city = loader.loadModel("city/city.X")
city.reparentTo(render)

# Get reference tocar factory instance
cf = carfactory.Instance

# Load buggy
# Ask the user which car he wants to drive
while 1:
    try:
        cartype = raw_input("What type of car do you want\
                                     to drive: ")
        aCar = cf.Create(cartype) # Use car object factory
        break
    except KeyError:
        print "Unknown type of car. Try again..."
```



The city and buggy example done right: Driving a car (cont.)

```
# Setup up the car in the scene graph
aCar.reparentTo(render)

# Setup camera frame (follow the buggy camera)
base.disableMouse() # Disable mouse controlled camera

# Choose lens type (Setup view frustum)
base.camLens.setNear(1)
base.camLens.setFar(10000)
base.camLens.setFov(50,40)

# Bind camera to car
aCar.bindCamera(base.camera)

# Set up event handling
evHandler = MyEventHandler(aCar)

# Game loop
run()
```



Scripting Programming Paradigm

The following paradigms or hybrids are interesting:

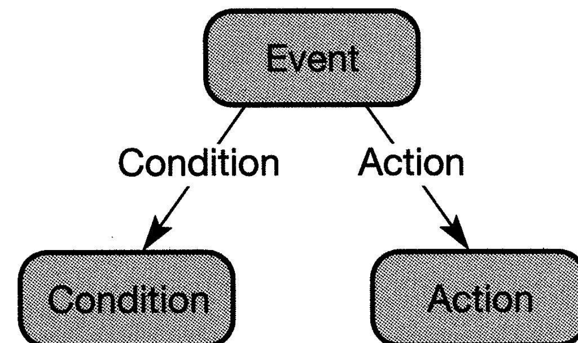
- Linear control
- Event-driven
- Prioritised task execution

Linear control:

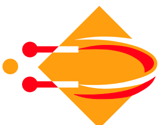
```
wait 10                     // Re-entrant: Takes time to execute
this.set_position(0,0,0) // Atomic: Executed immediately
this.set_velocity(0,0,1) // Atomic
wait_until(this.z >= 10) // Re-entrant function
```

Event-driven: The script consists of pieces of code for handling events.

```
event create
   this.energy = 1000
end_event

event destroy
   print "Bang!"
end_event
```



Prioritised task execution: The script consists of prioritised tasks that are executed in order until completion.

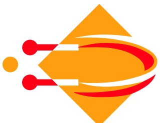


Scripting Programming Paradigm in Panda3d

Panda3d uses a hybrid between tasks and events.

Defining your own panda tasks:

```
from direct.task import Task

def myTask(task):
...

taskMgr.add(...)
taskMgr.Remove(...)
taskMgr.doMethodLater(<delay time>, ...)

return Task.cont
return Task.done
...
```



The Panda3d Game Loop

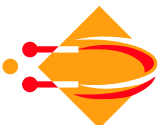
What happens in `run()`?

For every frame, the game loop executes every registered task in a prioritised order.

Some standard Panda tasks are:

```
eventManager  # Process events
igloop        # Draw the scene
collisionloop # Do collision detection
dataloop      # Processes the data graph
```

Lets consider an example.

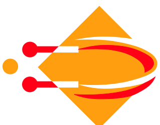


How is my animation performing? ⇒ Is my program efficient?

Animations should appear smooth. Smoothness is measured in how many images (frames) can be rendered per second (Frames per second [fps]). The human visual system requires at least 25 fps.

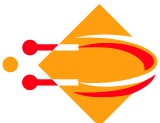
How to measure the frame rate in Panda:

```
from direct.task import Task
class myFrameRate:
  """myFrameRate:
     Sets up a task for printing the framerate to the console
     """
  last = 0
  fps = 0
  def __init__(self):
      taskMgr.add(self.myFrameRateTask, "myFrameRateTask")

  def myFrameRateTask(self,task):
    if (task.time - self.last)
                 > globalClock.getAverageFrameRateInterval():
        self.fps = globalClock.getAverageFrameRate()
        print "Fps = " + str(self.fps)
        self.last = task.time

    return Task.cont
```



Using my frame rate task

```
import direct.directbase.DirectStart # Start Panda
from framerate import myFrameRate

# ... Code for setting up your game ...

# Using my frame rate class
fps = myFrameRate()

print taskMgr # Lets see what tasks are running

run()
```



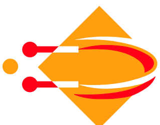
Representing Cars and Machine Guns: Game Objects

Definition of a *game object*:

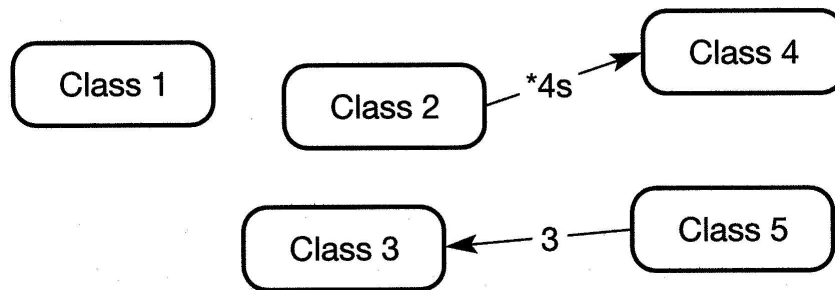
- In-game objects like cars and machine guns should be represented by a program object — *a game object*.
- Game objects can also represent abstract concepts like positions in your virtual world where events should happen whenever the player gets near. Example: The finishing line in your car game.

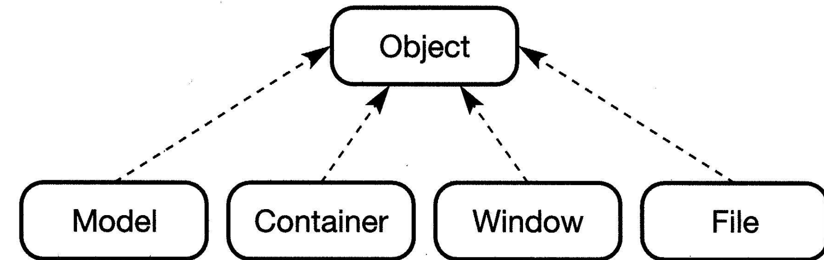
Gold [JG] considers the pros and cons of the following types of inheritance hierarchies for game objects:

- Collapsed
- Shallow
- Vertical
- Mix-in inheritance



Game objects: Collapsed and shallow



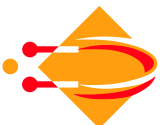


Pros and cons:

- Not really object oriented.
- You probably need to rethink your design and find common functionality of your game objects.

Pros and cons:

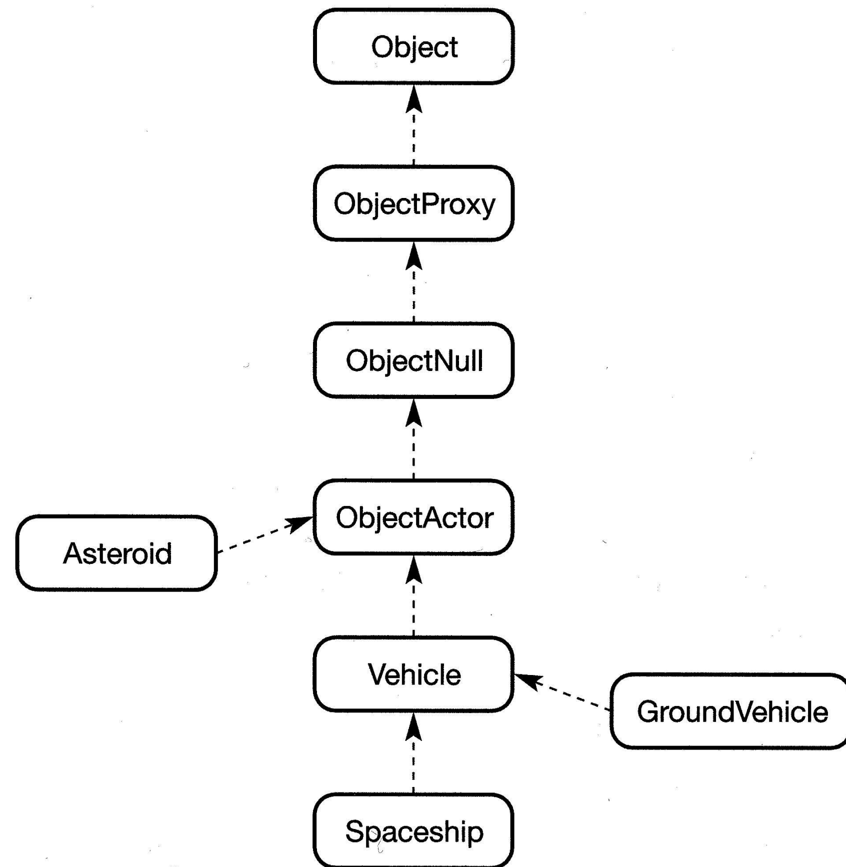
- Somewhat better. Collected common functionality in `Object`.
- But why does a container and a file share the same base class?
- Are there classes in your system that inherits functionality it does not need?

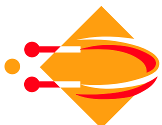


Game objects: Vertical

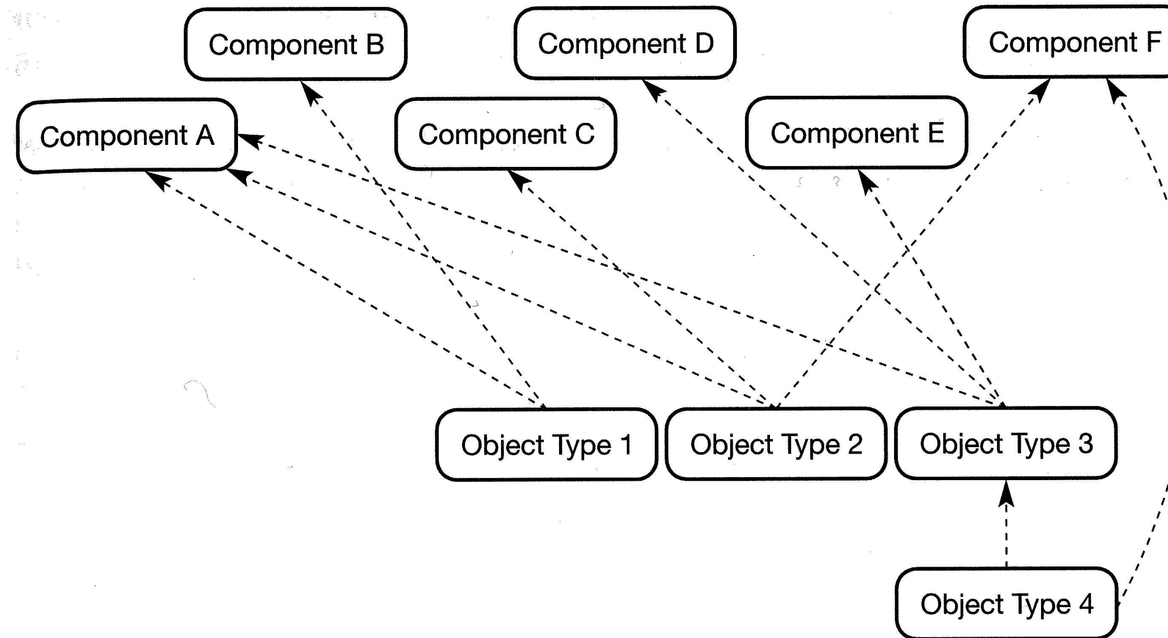
Pros and cons:

- Good that functionality is divided into smaller separate classes.
- Ease of maintainability.
- But making a change to `Object` will cause a recompilation of a lot of code.



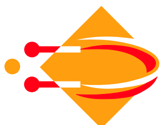


Game objects: Mix-in inheritance



Pros and cons:

- Mix-in the functionality you need for a specific class.
- The hierarchy is shallow, limiting the cost of making changes to base classes.
- Multiple inheritance can be somewhat tricky.



Gold [JG] recommends this inheritance hierarchy!

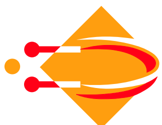
Game objects: Mix-in inheritance (cont.)

Example: The buggy class inherits from both the `car` interface and the `Actor` class:

```
class Buggy(car.Car,Actor.Actor):
...
```

In C++ care must be taken when using multiple inheritance:

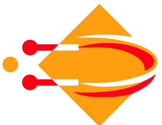
```
class Car {
  virtual void draw();
};

class Actor {
  virtual void draw();
};

class Buggy : public Car, public Actor {
  virtual void draw() {
    Car::draw();
    Actor::draw();
  }
};
```



Disambiguation of names needed. Avoid cyclic multiple inheritance — Classes should be orthogonal!

Overview of today

- Event handling
- Animation
- Script programming paradigm
- The game loop
- Game objects



Reading material

Reading material for this lecture:

- Parts of the Panda documentation (see course home page)
- Alan Watt and David Eberly: Hand-outs (you will get an e-mail about this)
- JG: Ch. 7.1, 8

Reading material for the next lecture:

- Alan Watt: Hand-out
- More to be announced