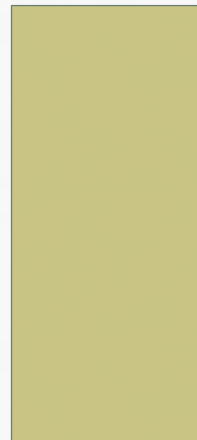


GAME ARCHITECTURE

THINK BIG
&
KEEP IT SIMPLE



CODE BY DESIGN

- Games are complex software
- Games are developed doing iterations
- Game are worked on by teams

So what can we do about it

UNIFIED MODELLING LANGUAGE

- Speak one diagram language
- Commonly known diagram styles
- Easy to understand if syntax known
- Provides quick insight, overview, details design
- Completeness does not mean understandable

DESIGN PATTERNS

- Use proven solutions, design patterns they solve problems you will encounter but have never heard of
- Design Patterns, Known solutions for Known problems
- Improved development, communication etc.
- Not the holy grail, but one needs stones to build

TEST DRIVEN DEVELOPMENT

- Programmers DO break thing, accept this and deal with it
- Need for (Integrity) testing and Fixing
- Blackbox (playtest), White box (review) testing....
- Invest in automated Testing.
Assertions as Guards, Unit Tests as Triggers
Before checking in code and avoid (nightly) build errors.
- Quality awareness is an attitude.

SIDEKICK THREADING

- Enable usage of multicores (*optional this year*)
- Risk of interference
 - Need for locking
 - Interproces locking using mutexes
 - Risk of deadlocks

EXAM

- Written and Assignment (50/50 > 6)
- Written :
 - Foundations on Linda : through act.saxion.nl :
 - Foundations of Programming: Object-Oriented Design
 - Foundations of Programming: Test-Driven Development
 - Patterns by Nystrom, Game Programming Patterns
 - UML : Use Case, Class, Sequence, State diagram.
- Assignment
 - Apply patterns as good design
 - Using Assertions and Unit tests for Code Quality
 - Have fun reengineering a simple game

A CUNNING PLAN

See manual.

GETTING STARTED

Who's Lynda.com

- Login through act.saxion.nl and your Saxion account.
- Thorough view on Object Oriented Design Principles

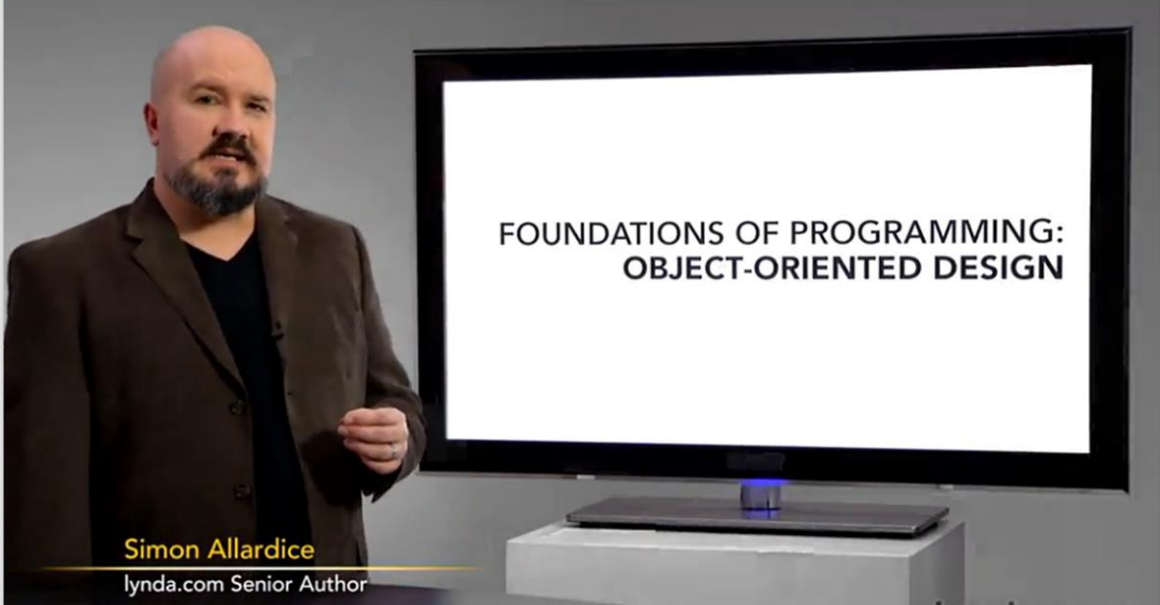
Online: Do it !!

≡ Browse the library Find courses, authors, and more... Search ≡ My courses

Foundations of Programming: Object-Oriented Design with Simon Allardice

Java Tutorials

In playlist ▾ Share Take a tour Use classic layout



Simon Allardice
lynda.com Senior Author

FOUNDATIONS OF PROGRAMMING:
OBJECT-ORIENTED DESIGN

Illustration by Mark Todd

LYNDA TO THE RESCUE

LYNDA.COM, LOGIN BY SAXION.NL ACCOUNT

CORE CONCEPTS

- Object
 - Class
 - Abstraction
 - Encapsulation
 - Inheritance
 - Polymorphism
-
- Accessors
 - Wrappers

OBJECT ORIENTED ANALYSIS & DESIGN

- Analyze and Design and Code
- Requirements
- Primary Design
- UML
 - Use Cases : user actions, events – from game design
 - Actors
 - Scenarios, including alternatives
 - Diagramming Use Cases
 - User Stories for development Scrumwise

DESIGN PRINCIPLES

- SOLID
 - Single Responsibility (clarity)
 - Open / Closed (extendable but stable)
 - Liskov Substitution Principle(implement to interfaces)
 - Interface Segregation Principle
(Decompose, client oriented interface)
 - Dependency Inversion Principle (abstract high to low)

Optional :

Linda, Learning S.O.L.I.D. Programming Principles

"You ain't gonna need it" rule from extreme programming (XP)(YAGNI)

DESIGN PRINCIPLES

- GRASP :
General Responsibility Assignment Software Patterns
 - Creator : who is responsible for creating a spec object
 - Controller : who is responsible for doing spec. job
 - Information Expert : if you have the data, you do the job.
 - Low Coupling : low dependency between classes
 - High Cohesion : strong cohesive set of responsibilities for classes.
 - Indirection : intermediate objects
 - Polymorphism : who's responsible for alternative behaviour
 - Protected Variations : handle changes using interfaces

DESIGN PATTERNS CLASSICS

- Nystrom: the Gang of Four Patterns
If it has a name, you may know all about it

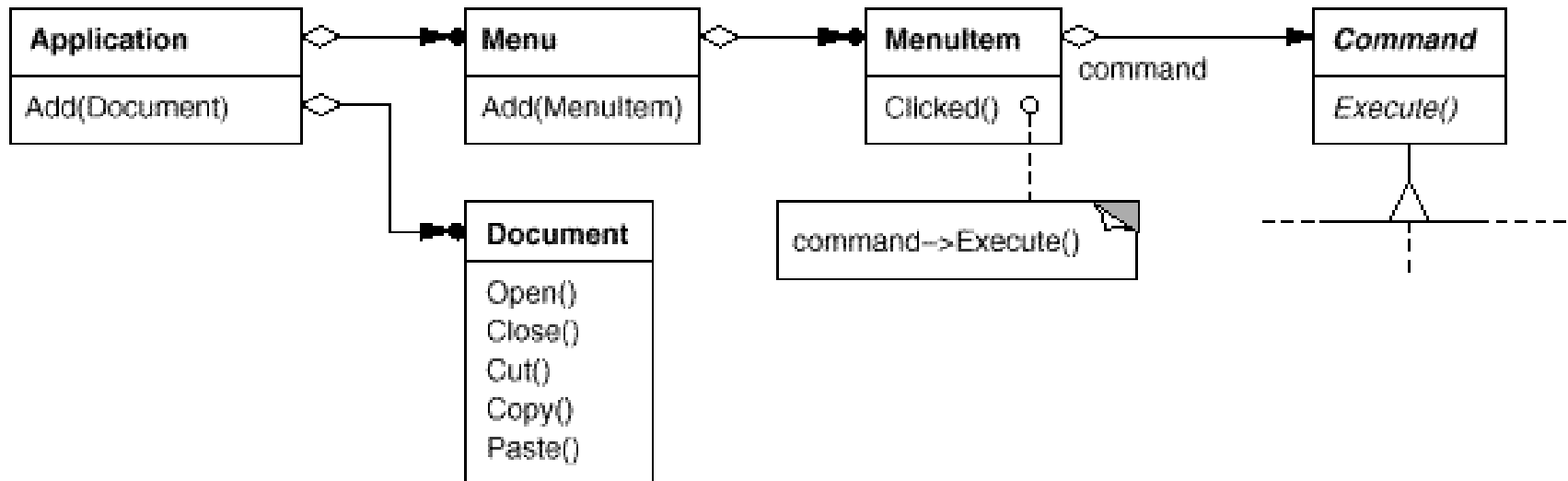
Classics : Behavioral, Creational, Structural (GoF),
1,2 Command,
4 Observer,
6 Singleton,
7 State

Gameprogramming Patterns : (Nystrom)
9 The Game Loop,
10 Update Method,
14 Component
15 Event Queue

There is a lot more, and also anti patterns....

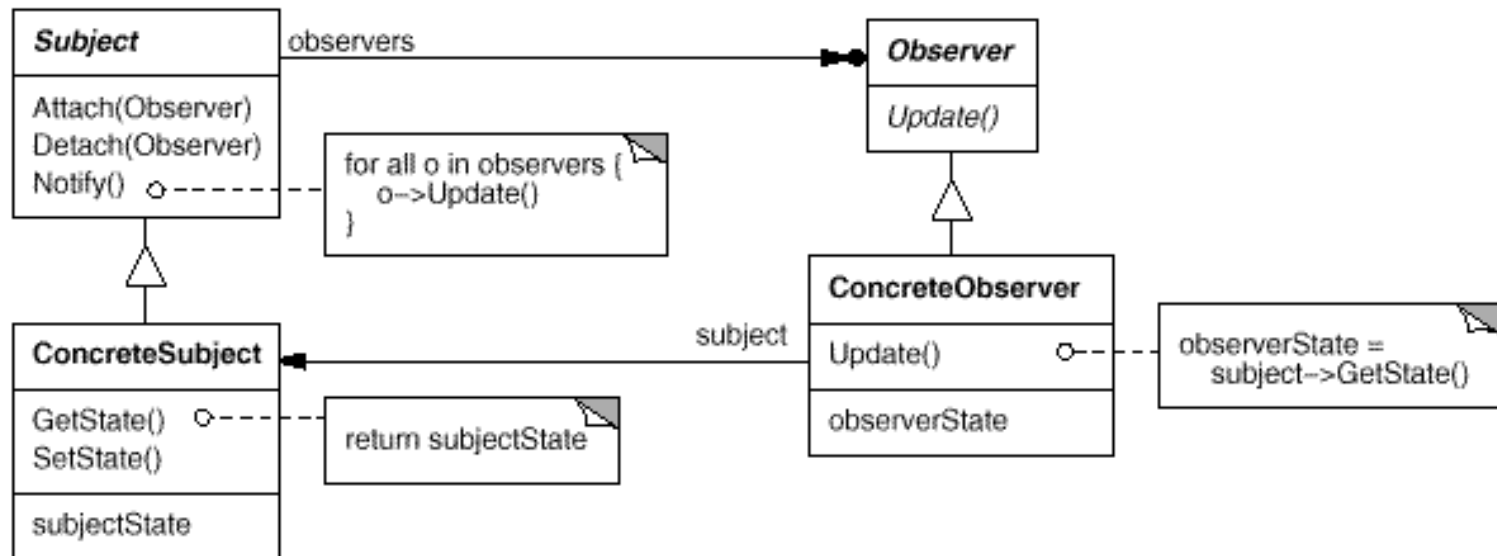
COMMAND PATTERN

- Behavioral Pattern aka Action
- Method call by object, decoupling action from source.
- Used for menu, queue, sequence or Undo/Redo



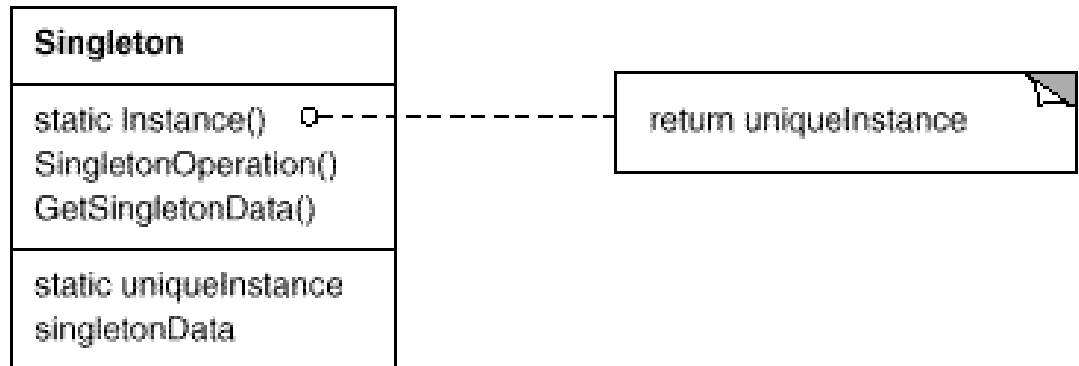
OBSERVER PATTERN

- Behavioral Pattern
- Get notified on Event
- Decouple sender and receiver for an event
- Core of MVC, Baked in c# as event



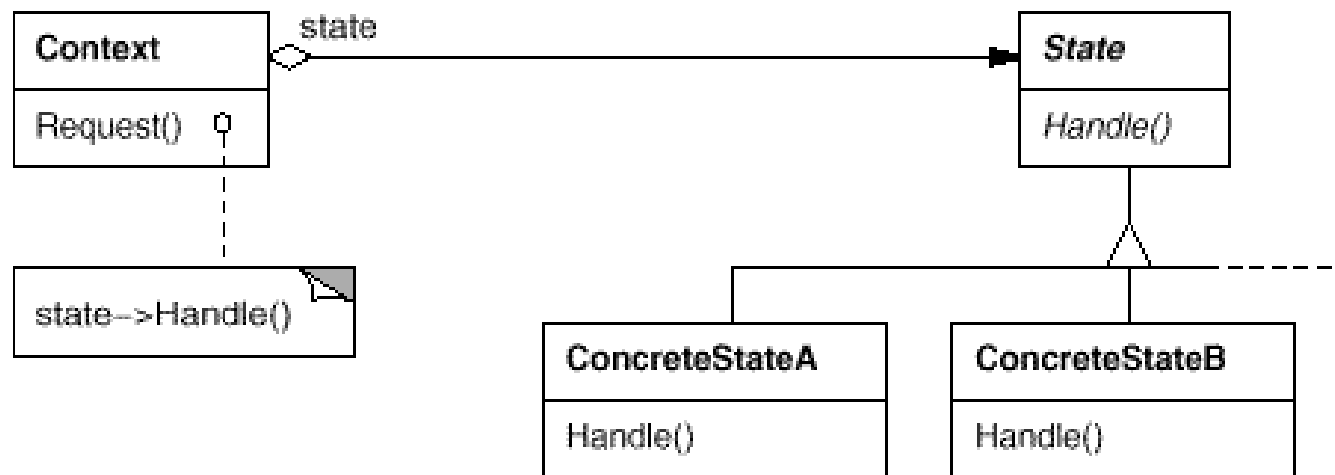
SINGLETON PATTERN

- “There can only be one”
 - Enforces singularity
 - Avoids globals or statics
- `Game.instance();` //the one and only
- Debated a lot.



STATE PATTERN

- Behavioral Pattern
- Actions depend on current state
- Used for AI: Idle, Running, Jumping, Hit, Dead.
- Often combined with message queues etc.



NYSTROM: GAMELOOP

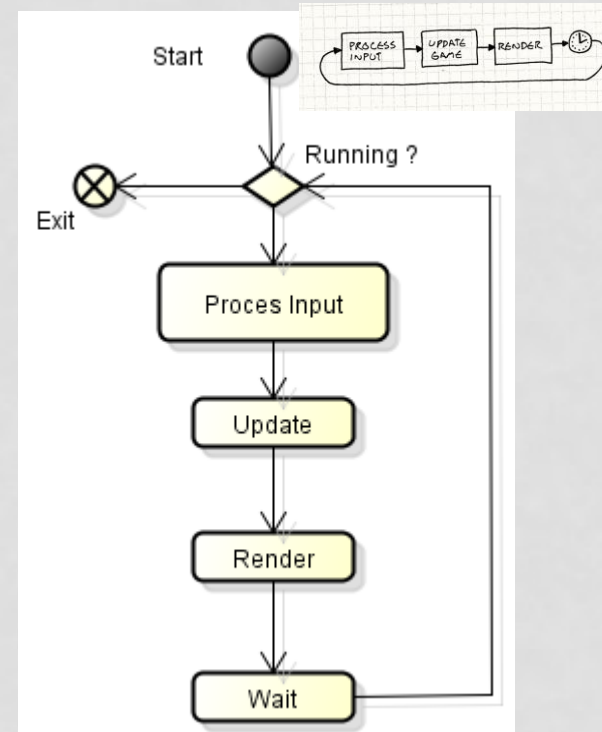
```
while( running ) { ... }
```

- continues simulation going on, even turn based
- handling user input by ‘event’ or ‘polling’

```
while( running ) {  
    ProcessInput(); // handling system events  
    Update(); // ai & physics  
    Render(); // show gamestate, hud etc..  
}
```

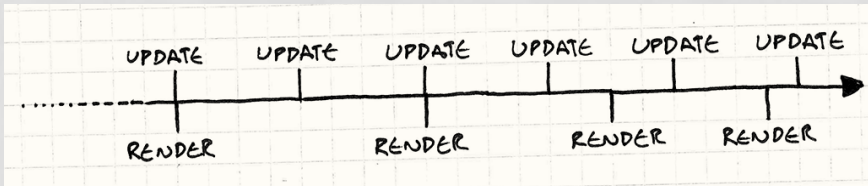
NYSTROM: GAMELOOP

- *What about time, step, tick or frame or ...*
- Frames per Second varies on workload and hardware
- Advance by dt ,
 - consistent over hardware
 - Problem
 - Non deterministic
- Better to catch up

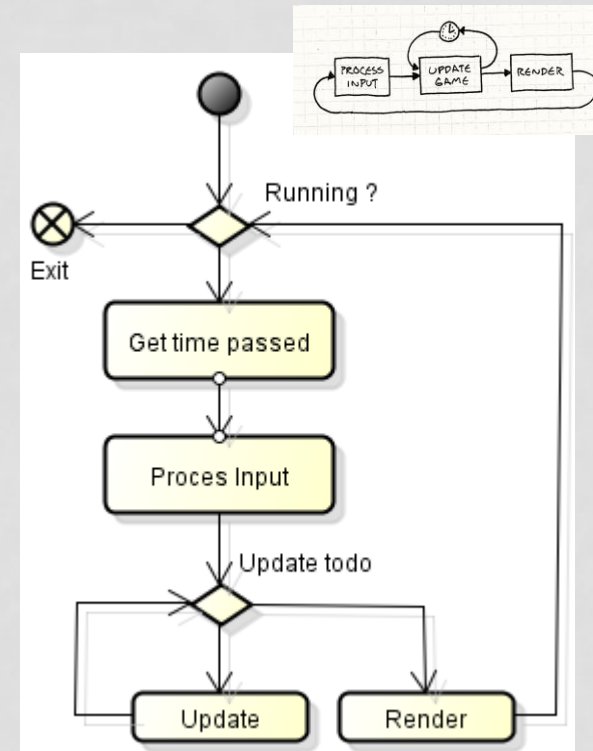


NYSTROM: GAMELOOP

- *Catching Up ?*
- Yes, do as many update as needed for time passed
 - Fixed steps, deterministic
 - Consistent over hardware
- There is more (problems)
 - Extrapolation for Render to overcome lag



http://docs.unity3d.com/uploads/Main/monobehaviour_flowchart.svg



NYSTROM: UPDATE

- *Can't use tight loops, need steps !*

You knew this ☺

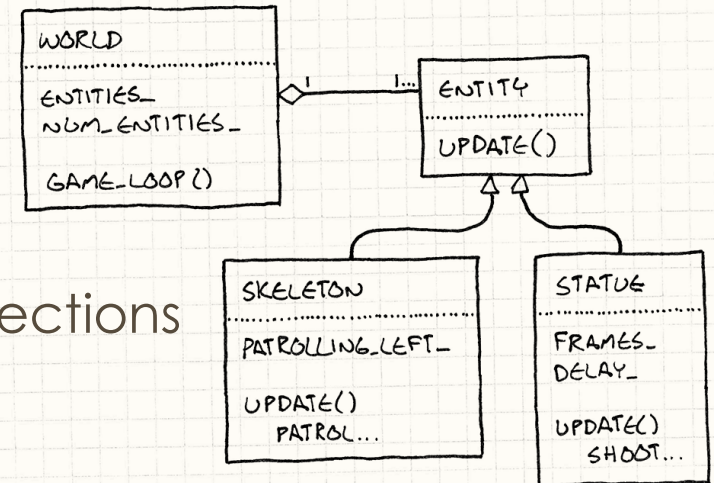
```
while (true)
{
    // Patrol right.
    for (double x = 0; x < 100; x++)
    {
        skeleton.setX(x);
    }

    // Patrol left.
    for (double x = 100; x > 0; x--)
    {
        skeleton.setX(x);
    }
}
```

- Simultaneous, independent behavior over time !

- Abstract Entity or GameObject class
- Enables the Component Pattern
- Enables the State Pattern

- Dormant objects: use separate collections



NYSTROM: COMPONENT

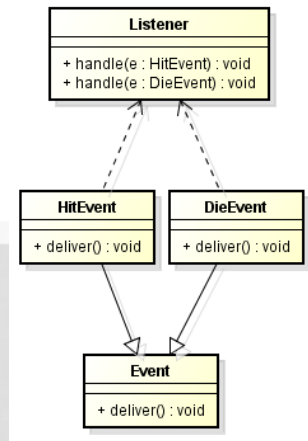
- Component Pattern
 - Remember Unity ?
 - Reusable components, runtime added
 - Flexibel : Plug and Play
 - Isolate Domains (Model, View, Controller ?)
- Inheritance
 - disables (swift) changes at runtime
 - many classes for many options
- Checkout [Bjorn](#)

C# ASSERTIONS

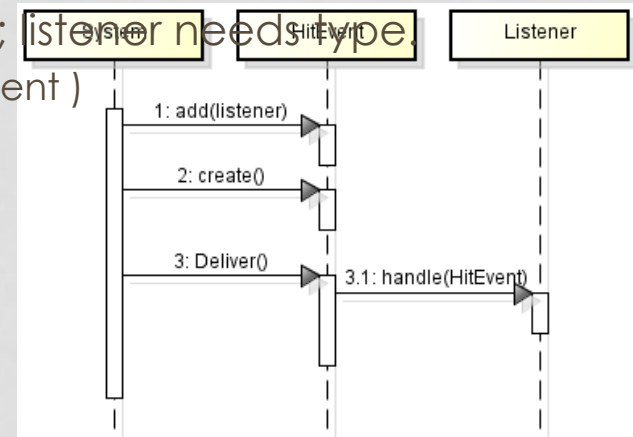
- Programmers inline check
 - Early error detection, white boxed.
 - Program should always be in valid state
 - Call should obey criteria (programming by contract)

ex. detecting null reference errors!!
- `System.Diagnostics.Debug.Assert(condition, message)`
- Included assertion tests in debug
- Excluded from Release
- Note : Assertions can be used in unittesting, but then have a different goal.

DOUBLE DISPATCHING



- *Optional, but handy.*
- How to know an subclasses type.
- Used in classis Visitor Pattern.
- I have two type of Event in the Queue, HitEvent and DieEvent, and they need to be delivered to Listener.
- Solutions...
 - Provide base event type, listener determines. (non scalable)
 - One function per type: `void handle(HitEvent e);`
 - Listener interface `handle(Event)` and `Handle(HitEvent)`
 - Have event deliver itself exposing it's type. (pseudo)
- https://en.wikipedia.org/wiki/Double_dispatch



QUESTIONS

?

UML



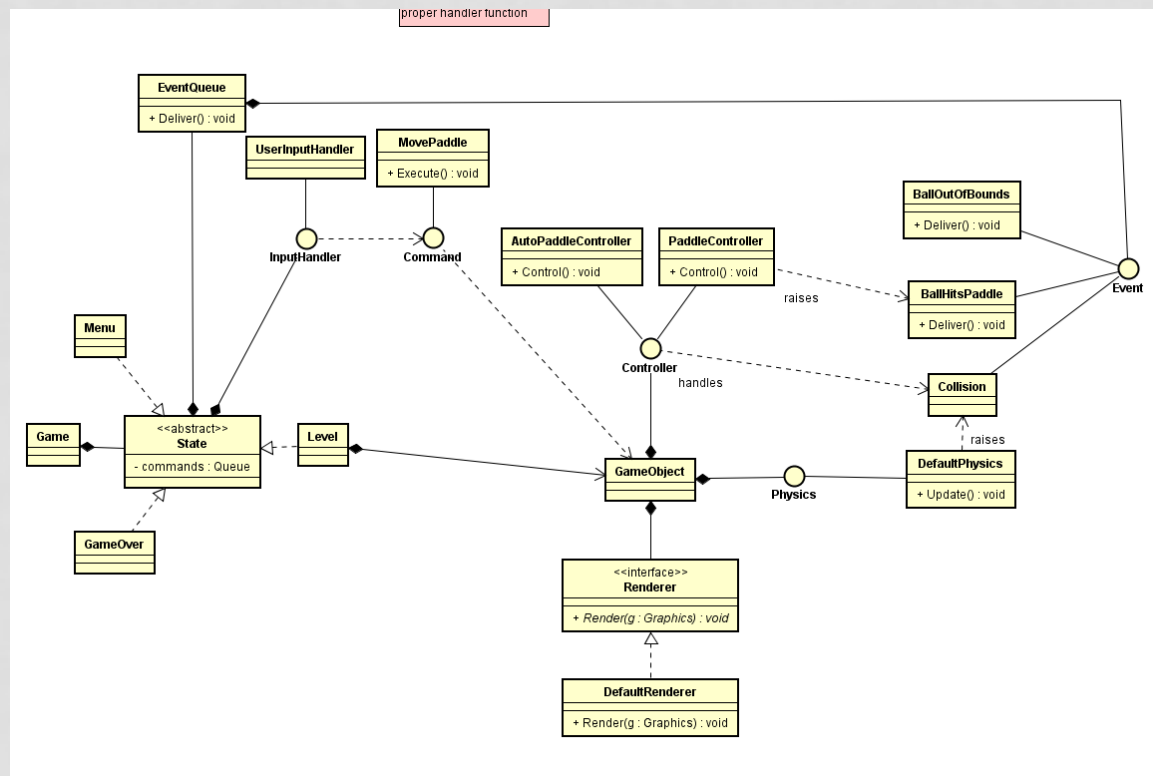
- Lets speak “Unified Modeling Language”
- Structure Diagrams
 - Class Diagram, etc..
- Behavior Diagrams
 - Use Case, Activity, State Machine Diagram, etc..
- Interaction Diagrams
 - Sequence Diagram

USECASE DIAGRAM

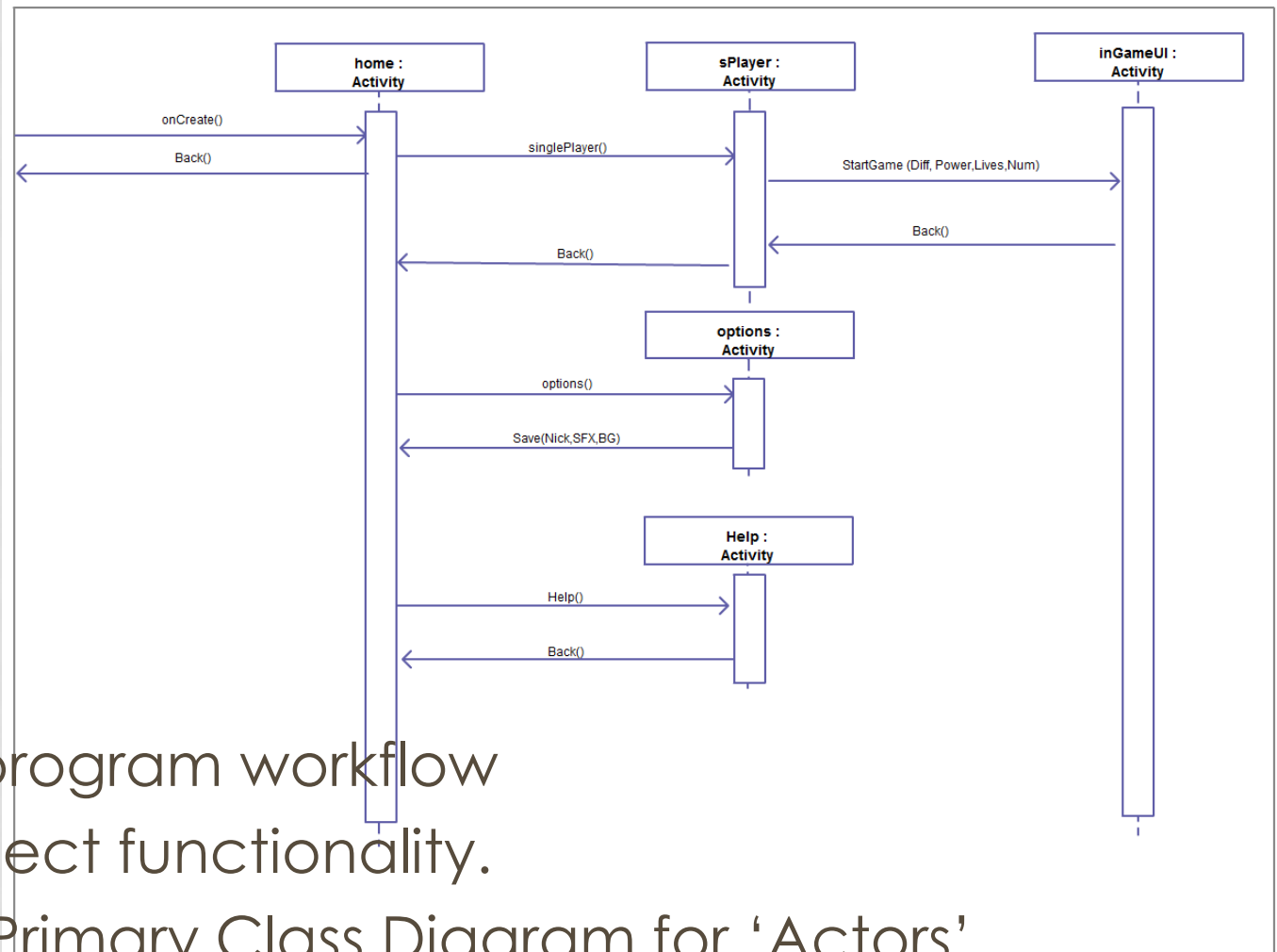
- Action on the system from 'actors'.
Actors are Users or subsystems (networking).
- Has include and extend
- 'Actor' perspective
- Pitfall: attempt to pseudocode.
- A simple game: start, control, pause and stop
- Chatserver : clients and master server

CLASS DIAGRAM

- For design and documentation.
- Has meaningful association and dependencies
- Can generate code skeleton



SEQUENCE DIAGRAM



- Discover program workflow
- Define object functionality.
- Start with Primary Class Diagram for 'Actors'

STATE DIAGRAM

- States and their Transitions (Enter, Exit)
 - Internal Guard
 - External Trigger
- Shows Behaviour
- Easy for communication with Game Designer too

ACTIVITY DIAGRAM

- Program flow, algorithm design (Flowchartish)
 - Starts
 - Action
 - Decision
 - Fork/Join
 - Comment
-
- Can be used for Gameflow too

WEEK 3

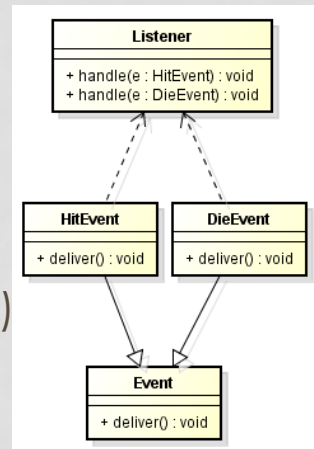
- Event Queue
- Assertions
- Unit Testing (Nunit) (opt)

EVENT QUEUE

- OS communicates with Application by Events
 - Application.Idle -> Handler (Object Sender, EventArgs e)
 - Form.Paint -> Handler(Object Sender, PaintEventArgs e)
- Have to register first: Application.Idle += Handler;
- Uses delegates, wrapped in EventHandler<EventType>
- Note: needs .net > 4.5, otherwise EventArgs

Handling and Raising events :

[https://msdn.microsoft.com/en-us/library/edzhd2t\(v=vs.110\)](https://msdn.microsoft.com/en-us/library/edzhd2t(v=vs.110))



EVENT QUEUE

```
class TimeoutEvent{
    // no interesting content here
}

class Timer {
    public event EventHandler<TimeoutEvent> timeout;

    public Timer( float interval ) {
        this.interval = interval;
    }
    public void Update( step ) { // somehow
        this.interval -= step;
        if( interval <= 0.0f ) {
            if( timeout != null ) {
                TimeoutEvent event = new TimeoutEvent();
                timeout( this, event ); // raise event
            }
        }
    }
}
```

//just example implementation, has issues

```
class User {
    private Timer;

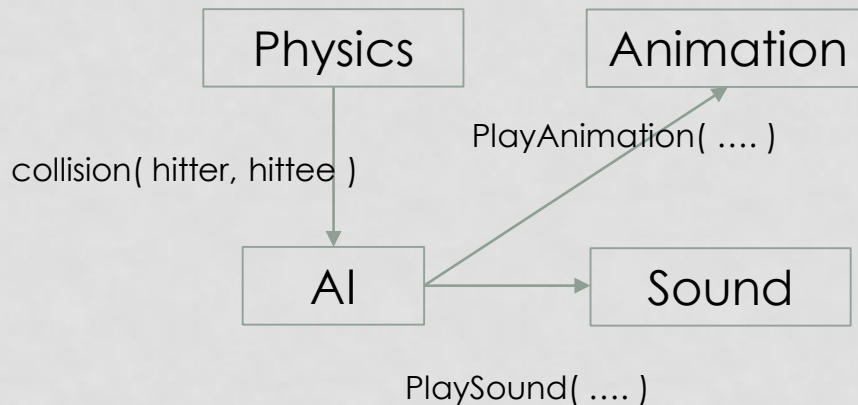
    public User() {
        timer = new Timer( 1000.5f );
        timer.timeout += Timeout;
    }

    public void Timeout( Object sender, TimeoutEvent e ) {
        // do interesting stuff
    }
}
```

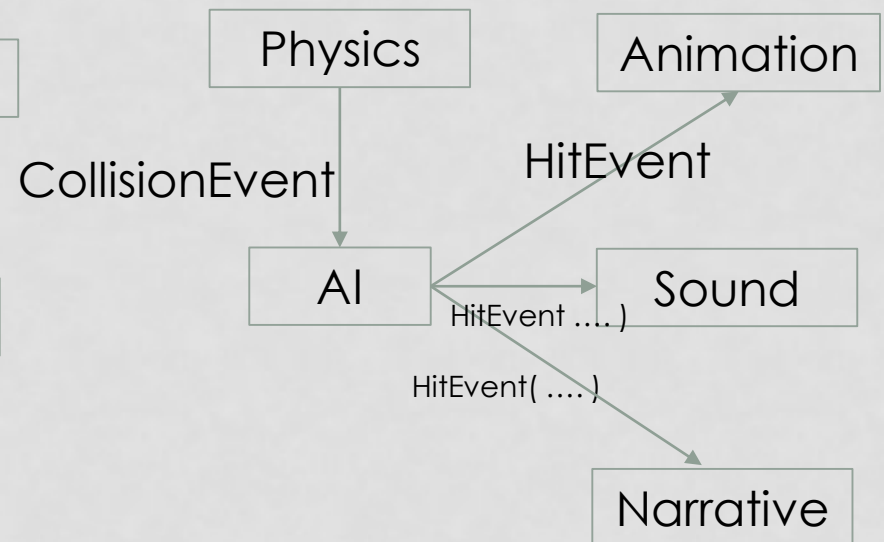
EVENT QUEUE

Decouple subsystems in the game engine

- Independent handling of events in system. (play sound on hit)
 - Act in your own time
 - Don't wait for others to finish
 - Potentially use Threads



By Call, tightly coupled



By Event, decoupled

EVENT QUEUE

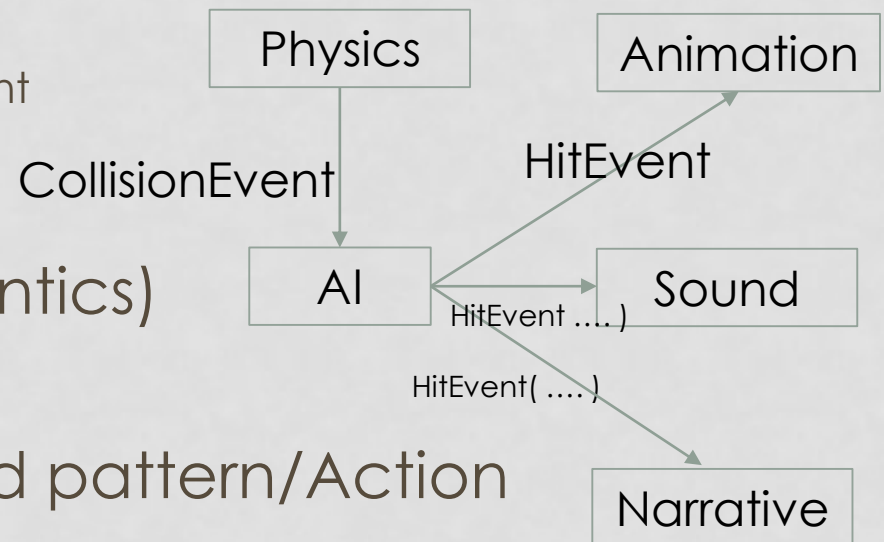
Store events in que for handling decouple in time

- Handler puts received Event in queue (sender thread)
- In loop/update System pumps the queue (or one at the time)

- Efficient queue: round robin
- C# has `System.Collections.Concurrent`

- Messages vs Events (symantics)

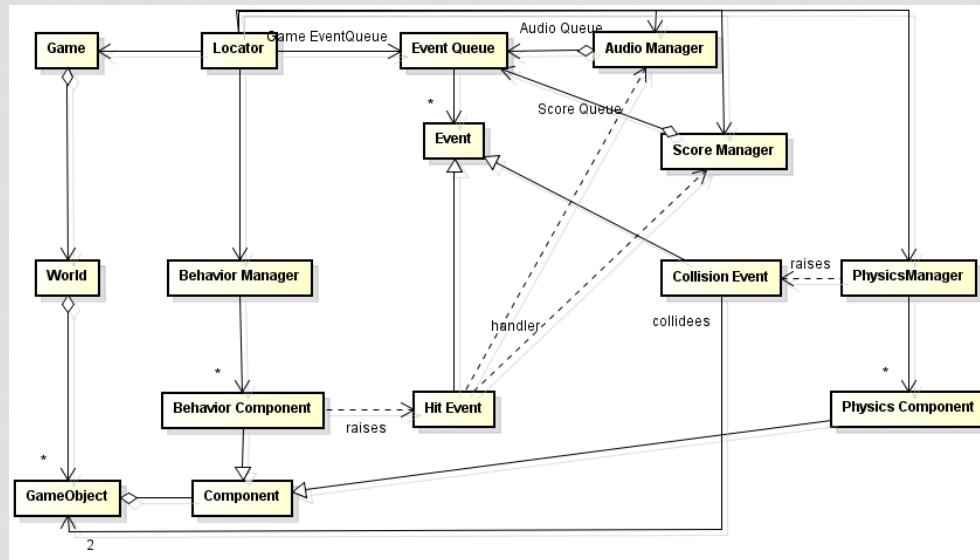
- Potential use of Command pattern/Action



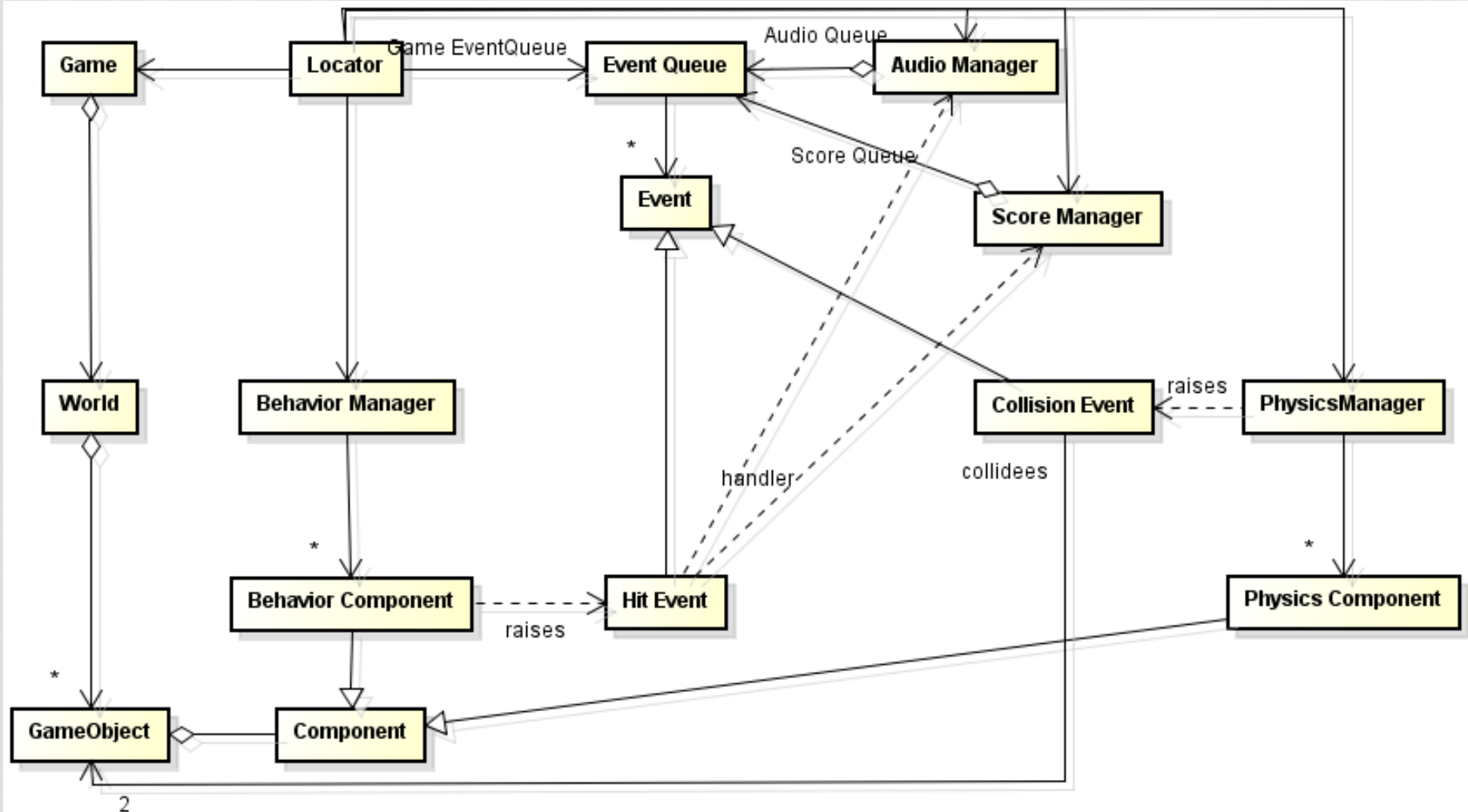
By Event, decoupled

EVENT QUEUE

An example Event System walkthrough, please note, variations are possible. (larger on next slide)

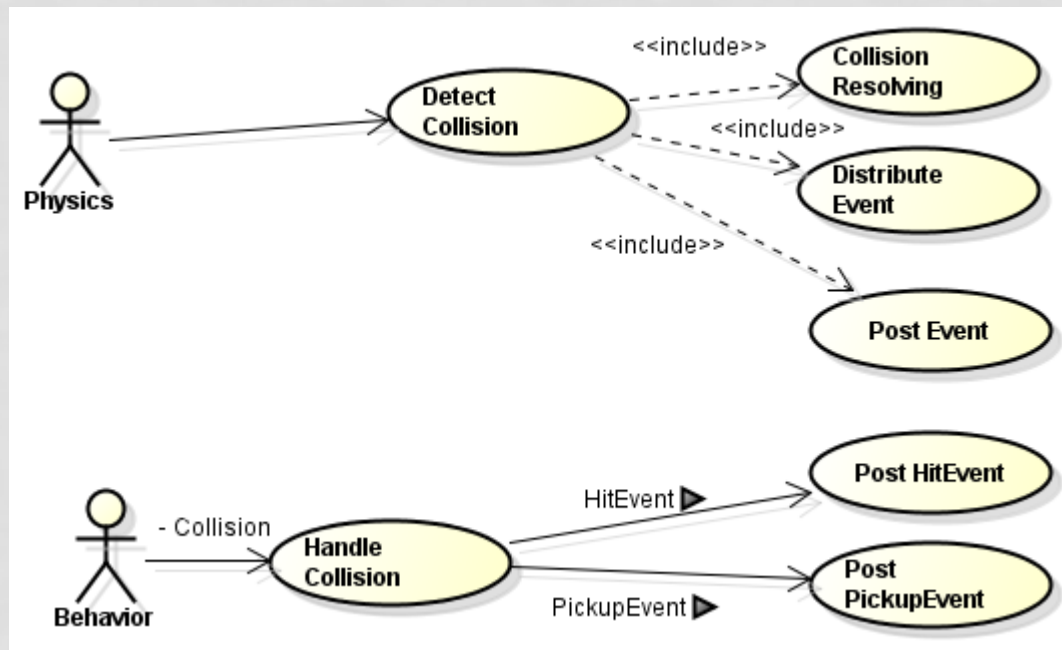


EVENT QUEUE



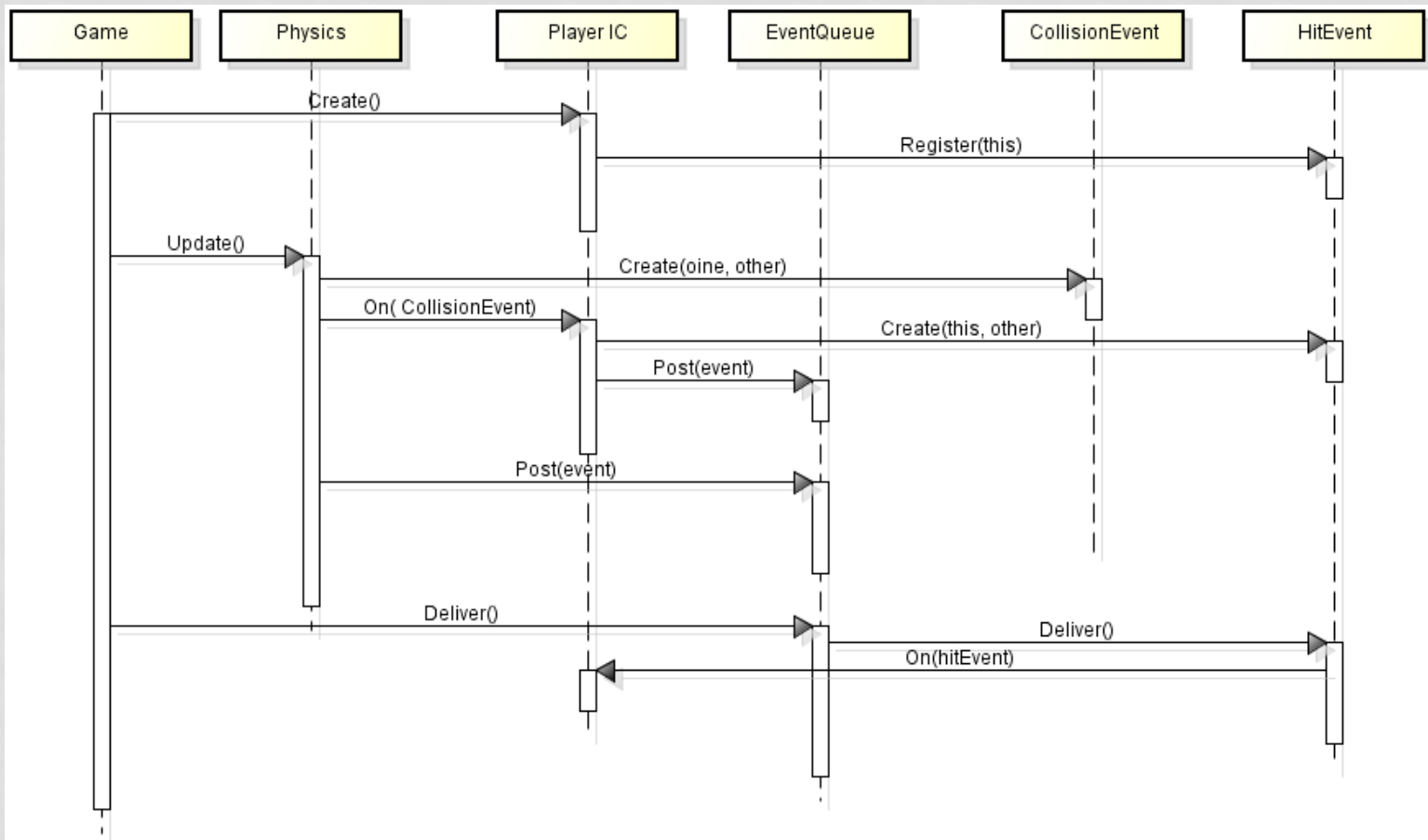
EVENT QUEUE

Use Cases (events example for subsystems)

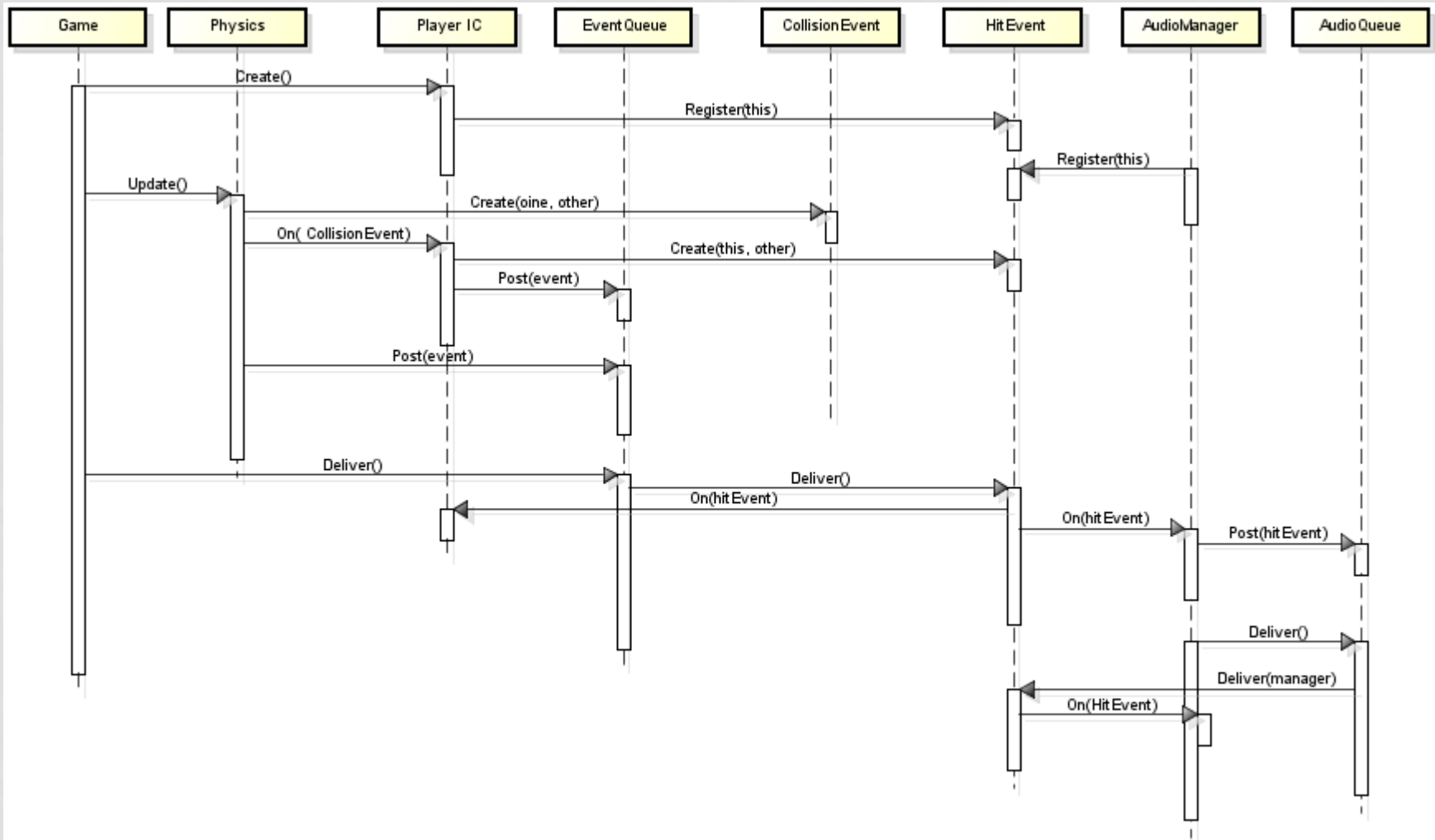


EVENT QUEUE

Sequence diagram (events example for subsystems)



EVENT QUEUE



QUESTIONS

?

MULTITHREADING

Optional

- Teamwork for Computers
- Use multicores in CPU,
requires Parallel Programming
- About Processes, Threads (and Tasks)

MULTITHREADING

Processes:

- Simultaneous execution of Programs on OS
OS distributes over cores or uses timeslices.
- Not sharing data, but sharing io
- Communicate through Sockets

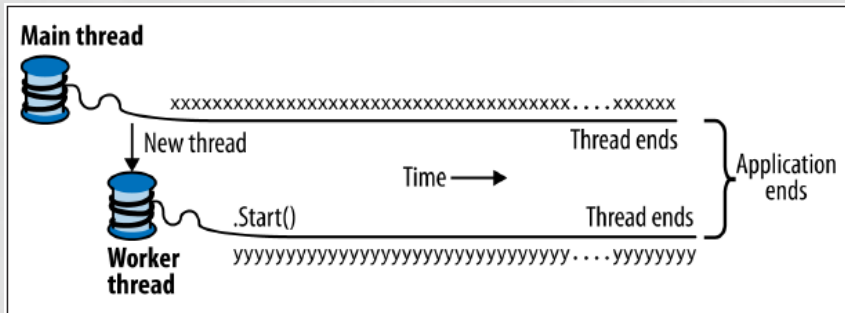
MULTITHREADING

Threads:

- Simultaneous execution of subPrograms.
Every process has at least it's main thread
- Sharing data.
- Problems
Race Conditions
Thread Safety

MULTITHREADING

Creating a Thread



```
using System;
using System.Threading;

class Program
{
    public static void Main(string[] args)
    {
        Thread thread = new Thread( Run );
        thread.Start();

        for( int i = 0; i < 1000; i++ ) {
            Console.Write("+");
        }

        Console.ReadKey(true);
    }

    public static void Run()
    {
        for( int i = 0; i < 1000; i++ ) {
            Console.Write("-");
        }
    }
}
```


MULTITHREADING

Sleep, Join

Blocking

vs

Spinning/Polling

```
//busy waiting, spinning  
while( a.busy() ) { };
```

```
// Blocking, no cpu  
thread.Jion();
```

```
using System;  
using System.Threading;  
  
class Program  
{  
    public static void Main(string[] args)  
    {  
        Thread thread = new Thread( Run );  
        thread.Start();  
  
        for( int i = 0; i < 1000; i++ ) {  
            Console.Write("+");  
        }  
        thread.Join(); // blocking  
  
        Console.ReadKey(true);  
    }  
  
    public static void Run()  
    {  
        for( int i = 0; i < 1000; i++ ) {  
            Console.Write("-");  
        }  
        Thread.Sleep( 100 ); // blocking  
    }  
}
```

MULTITHREADING

Share Data

Risk of interference
non Atomic Operation

lock to protect
or
use Interlocked class
for atomic operations

```
using System;
using System.Threading;

class Program {
    static readonly Object key= new object();

    public static void Main(string[] args) {
        Program program = new Program();

        Thread thread = new Thread( program.Run );
        thread.Start();

        lock(key) { // wait for locker to be free
            for( int i = 0; i < 2000; i++ ) {
                Console.Write("+");
            }
        }
        thread.Join(); // wait for worker to end;
    }

    public void Run() {
        lock(key) { // wait for locker to be free
            for( int i = 0; i < 1000; i++ ) Console.Write("-");
        }
    }
}
```

MULTITHREADING

Priority

- differs per OS
- handle with care
can freeze others

```
using System;
using System.Threading;

class Program
{
    public static void Main(string[] args)
    {
        Thread thread = new Thread( Run );
        thread.Priority(ThreadPriority.Lowest );
        thread.Start();

        for( int i = 0; i < 1000; i++ ) {
            Console.Write("+");
        }
        thread.Join(); // blocking

        Console.ReadKey(true);
    }

    public static void Run()
    {
        for( int i = 0; i < 1000; i++ ) {
            Console.Write("-");
        }
        Thread.Sleep( 100 ); // blocking
    }
}
```

MULTITHREADING

Synchronizing threads

- Use Signals, blocking

```
using System;
using System.Threading;

class Program {
    readonly Object locker = new object(); // protects Console.Write
    EventWaitHandle signal = new AutoResetEvent( false ); // true blocks
    Thread thread;

    public static void Main(string[] args) {
        Program program = new Program();

        program.thread.Start();

        for( int i = 0; i < 1000; i++ ) {
            Console.Write("+");
        }

        program.thread.Join();

        Console.ReadKey(true);
    }

    public Program() {
        thread = new Thread( Run );
        thread.Name = "Worker";
    }

    public void Run() {
        signal.WaitOne( 1000 ); // wait for signal to be set or timeout
        for( int i = 0; i < 1000; i++ ) {
            Console.Write("-");
        }
    }
}
```

MULTITHREADING

Thread pool

- Overcome creational overhead for Stack creation etc

- C# has Threadpool for short running threads. like Tasks, Timer

- With Result value Task<int>

```
using System;
using System.Threading.Tasks;

class Program {
    public static void Main(string[] args) {
        Program program = new Program();

        Task task = Task.Run( () => { // lambda or inline function
            for( int i = 0; i < 1000; i++ ) {
                Console.Write("-");
            }
        });

        for( int i = 0; i < 1000; i++ ) {
            Console.Write("+");
        }

        Console.ReadKey(true);
    }
}
```

MULTITHREADING

Multithreaded Game Engine

- Race Conditions: requires Threadsafety (read/write)
- Window operations within own creating thread

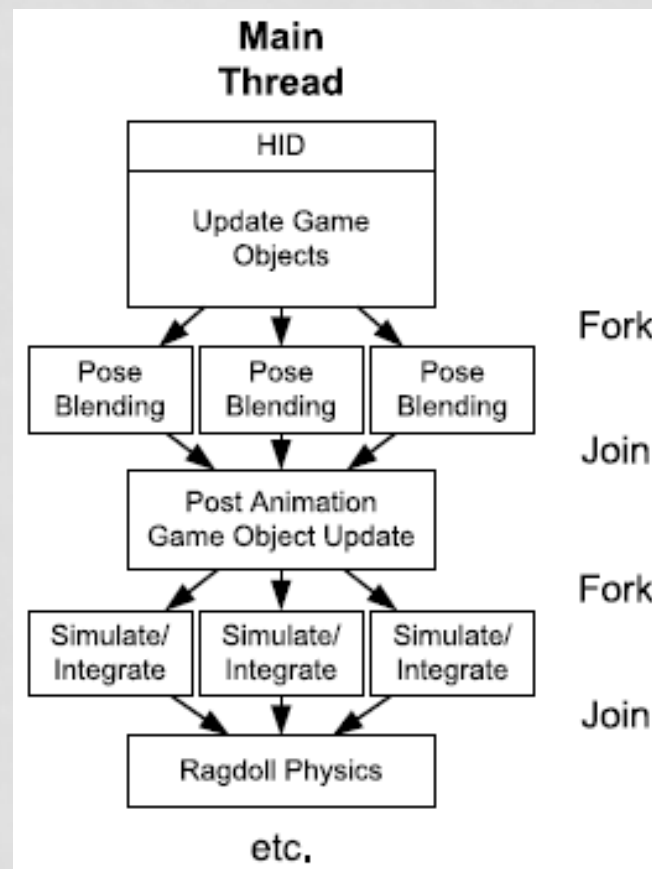
Different Designs (Gregory)

- Fork and Join
 - One Thread per Subsystem
 - Jobs
-
- All avoid read/write parallel, difficult to design

MULTITHREADING

Multithreaded Game Engine

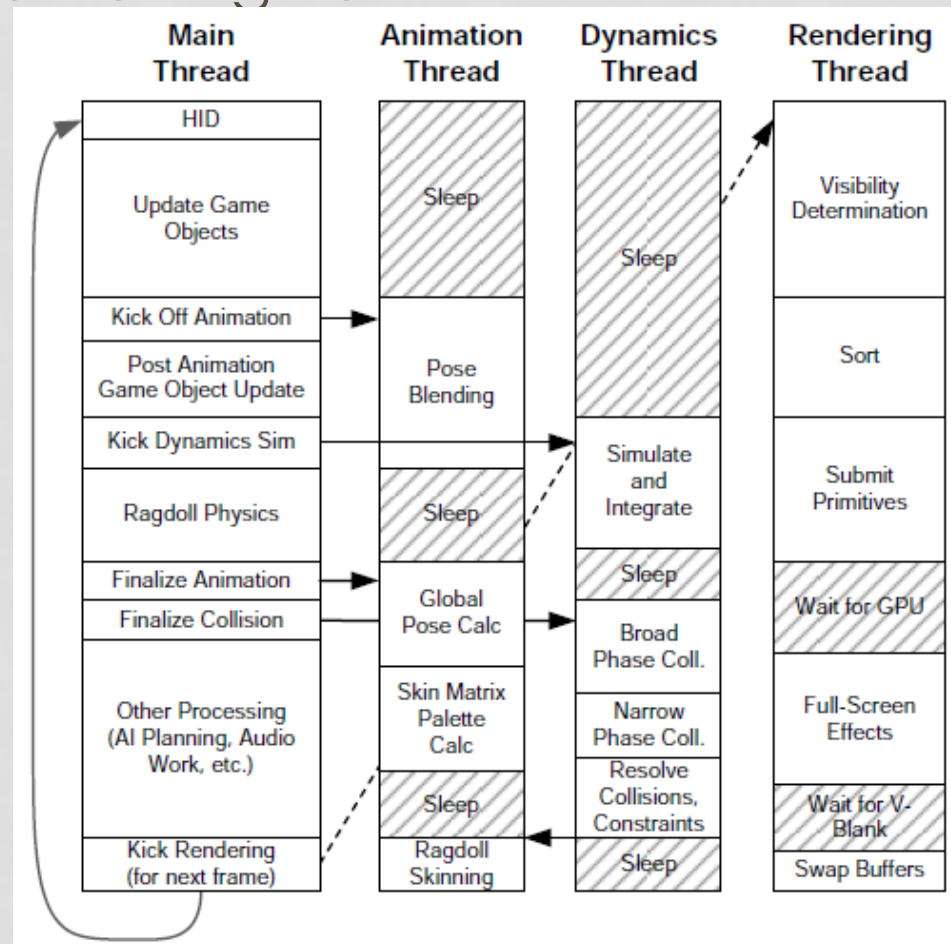
- Fork and Join



MULTITHREADING

Multithreaded Game Engine

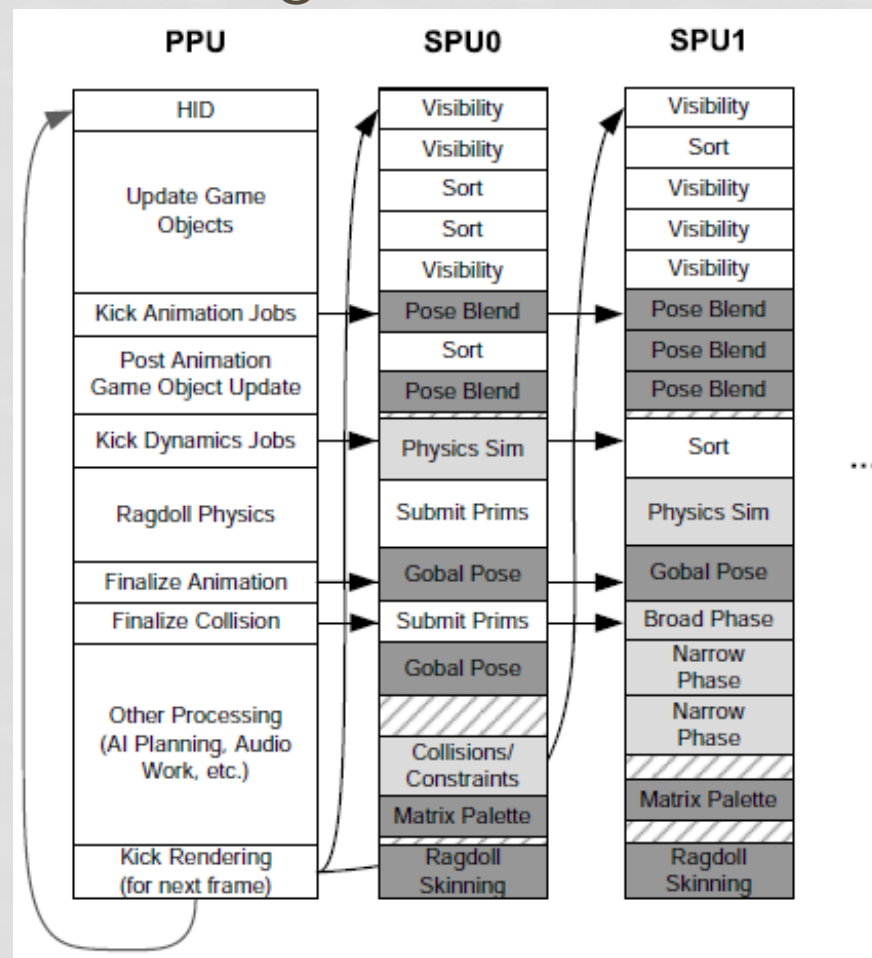
- One Thread per Subsystem



MULTITHREADING

Multithreaded Game Engine

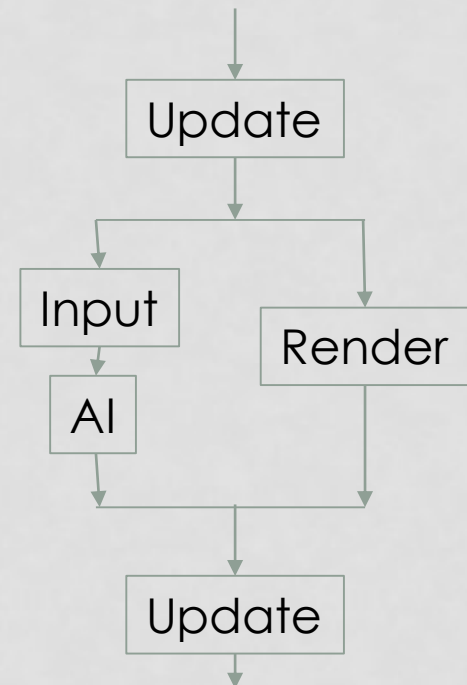
- Jobs



MULTITHREADING

Apparently parallel running jobs

- True on multicore (Reason to put up with the problems)
- Time slicing on single (preemptive or cooperative).
- Working with priorities
- Lock to protect shared resource
- Start thread to fork, end loop to finish
- Wait to synchronize, avoid restart
WaitHandle.WaitOne
- Join for Thread end
- C# has Task using pool.



QUESTIONS

?

QUALITY

- Assertions
- Unittesting

QUALITY

Assertions

- System.Diagnostics
 - Debug.Assert(bool, string)
 - Active in debug, inactive in release

```
public void DoSome( Object o ) {  
    Debug.Assert( o != null, "Object cannot be null !" );  
}
```

- Note: Unity has own (more finegrained).

QUALITY

Unit Testing

- Framework Nunit (or other)
 - Adds Assert class with Fail etc...
 - Separated Test classes
 - Multiple independent Test methods on public members
 - Run All

QUALITY

Unit Testing class (using NUnit.Framework)

```
[TestFixture]
public class NUnitTest
{

    [Test]
    public void TestIsValid()
    {
        MyClass my = new MyClass();
        my.value = -1;
        if( my.isValid() ) {
            Assert.Fail( "isValid is faulty" );
        }
    }
}
```

QUESTIONS

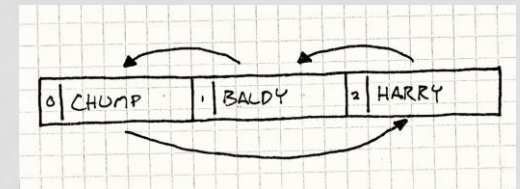
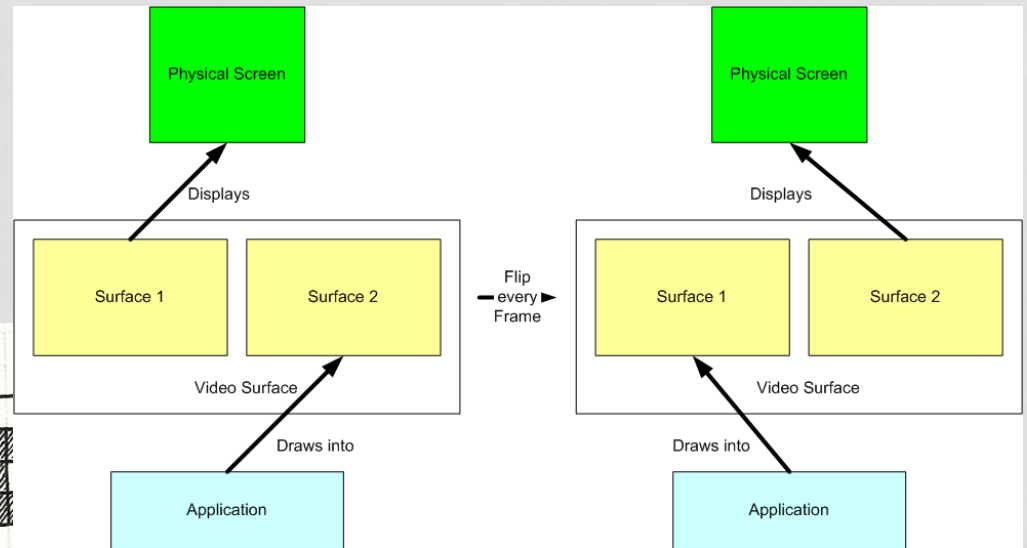
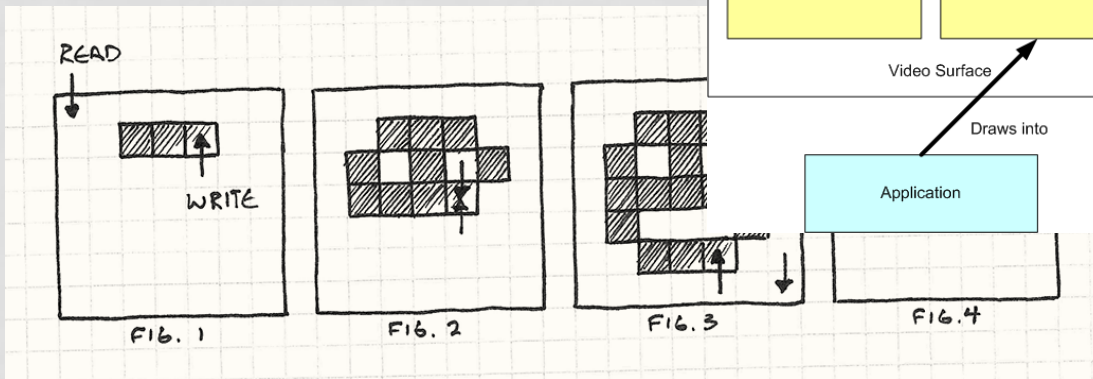
?

DESSERT

- Double Buffer
- Object Pool
- Space Partitioning

DESSERT

- Double Buffer tearing swap



- Applicable for Multithreading (pos) !

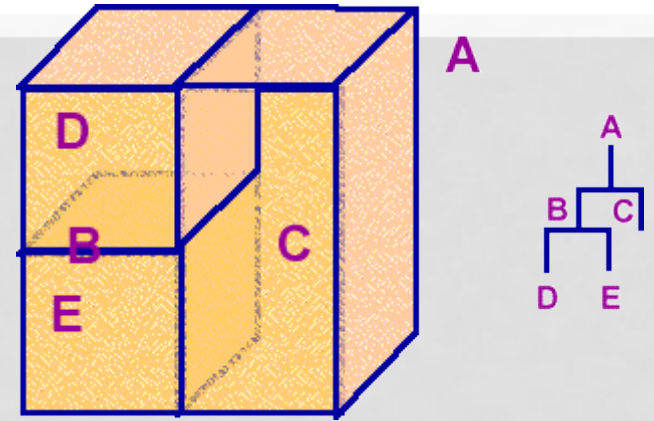
DESSERT

- Object Pool
- Seen with Thread Pool
- Avoid continuously create destroy overhead.
New/Delete is expensive
- Reuse objects in Pool
Some overhead for available, no new/del
Needs max count
Possibly optimizing mem caching

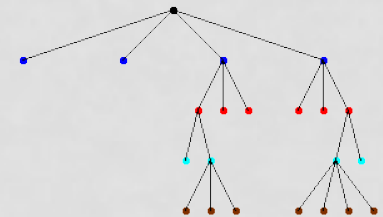
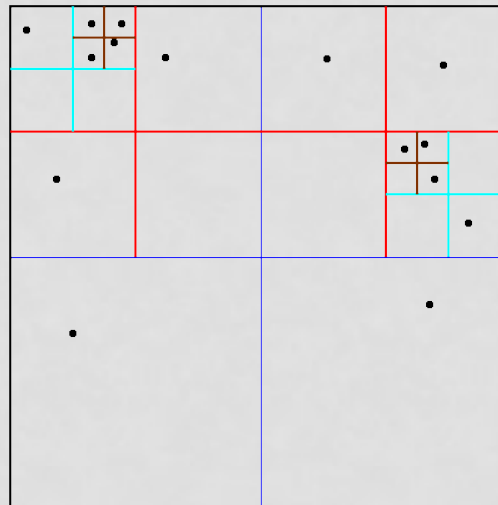
DESSERT

Space Partitioning

- Reduce collision pairs early
- Group in reduced space
- Octree is often used
- Static/Dynamic objects.



Adaptive quadtree where no square contains more than 1 particle



QUESTIONS

?

EXAM

Lab assignment

Demonstrating UML

UseCase, Class, State, Sequence (optionally more, like Activity)
diagramm

Applying selected Patterns (Based on Nystrom)

Gameloop, Update, Component, State and Command, Locator

Applying Assertions and optionally UnitTesting

Exam

- Lynda.com : Foundations of Programming: Object-Oriented Design
- Lynda.com : Foundations of Programming: Test-Driven Development.
- Selected Nystrom Patterns Details

EXAM

1. What is the order of the five steps to identify the classes : (give best)

- a. Describe, Identify, Class Diagram, Requirements, Interactions
- b. Requirements, Interactions, Identify, Describe, Class Diagram
- c. Describe, Requirements, Interactions, Class Diagram, Identify
- d. **Requirements, Describe, Identify, Interactions, Class Diagram**

2. What is NOT a group of Design Patterns

- a. Creational Patterns
- b. Structural Patterns
- c. Behavioral Patterns
- d. **Consistency Patterns**

3. What is NOT typical for Unit Testing

- a. Have separated Test classes
- b. **Check for unwanted states**
- c. Have independent Test methods
- d. Present a collection of runnable tests