Game Architecture

4/5/16: Time

Periodic Servicing

- The rates in which subsystems need to be serviced varies
- Animation system typically synced with rendering system
- Dynamic simulation might need to be updated twice as often (~120Hz)
- AI might only need to be serviced twice a second

The Windows Message Pump

```
while (true)
{
  //Service any and all pending Windows messages

  MSG msg;

  while (PeekMessage(&msg, NULL, 0 ,0) > 0)
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  //no more messages to process
  RunGameLoopOnce();
}
```

Frameworks

```
while (true)
{
  for (each frameListener)
  {
    frameListener.frameStarted();
  }

  renderCurrentScene();

  for (each frameListener)
  {
    frameListener.frameEnded();
  }

  finalizeSceneAndSwapBuffers();
}
```

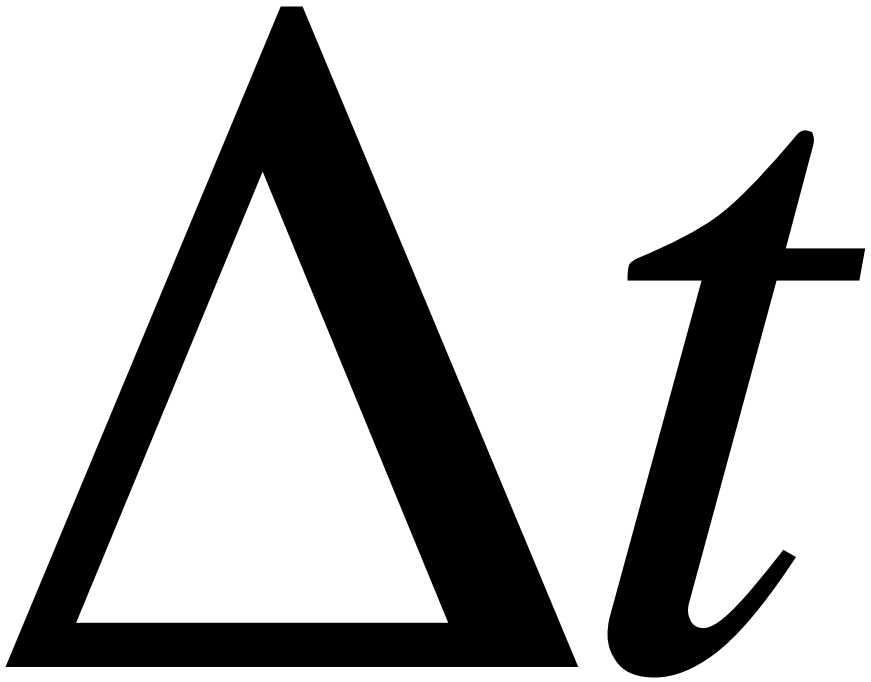
Frameworks

```
class GameFrameListener : public Ogre::FrameListener
{
public:
  virtual void frameStarted(const FrameEvent& event)
  {
    //Do all things that must happen before rendering
    pollGamepad(event);
    updatePlayerControls(event);
    updateDynamicsSimulation(event);
    resolveCollisions(event);
    updateCamera(event);
    //etc...
  }

  virtual void frameEnded(const FrameEvent& event)
  {
    //do things that happen after rendering
    drawHud(event);
    //etc...
  }
}
```

Event-Based Updating

- Event systems allow subsystems to register "interest" in particular kinds of events and to respond when they occur
- Engines often leverage event systems to implement periodic servicing
  - System must allow events to be posted into the future

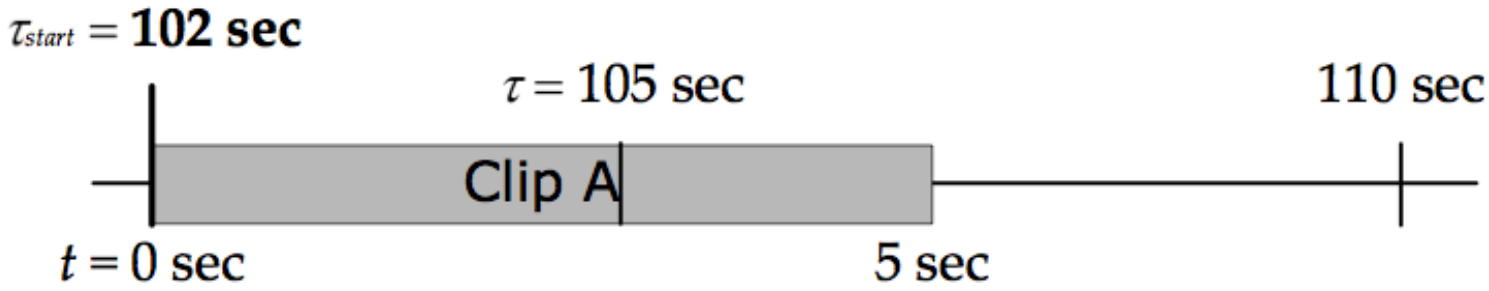


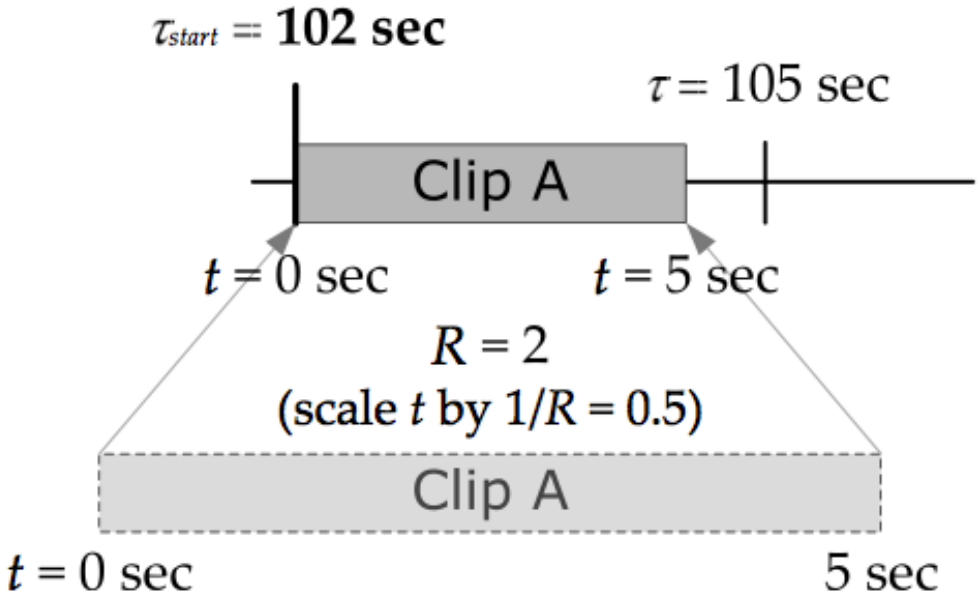
Abstract Timelines

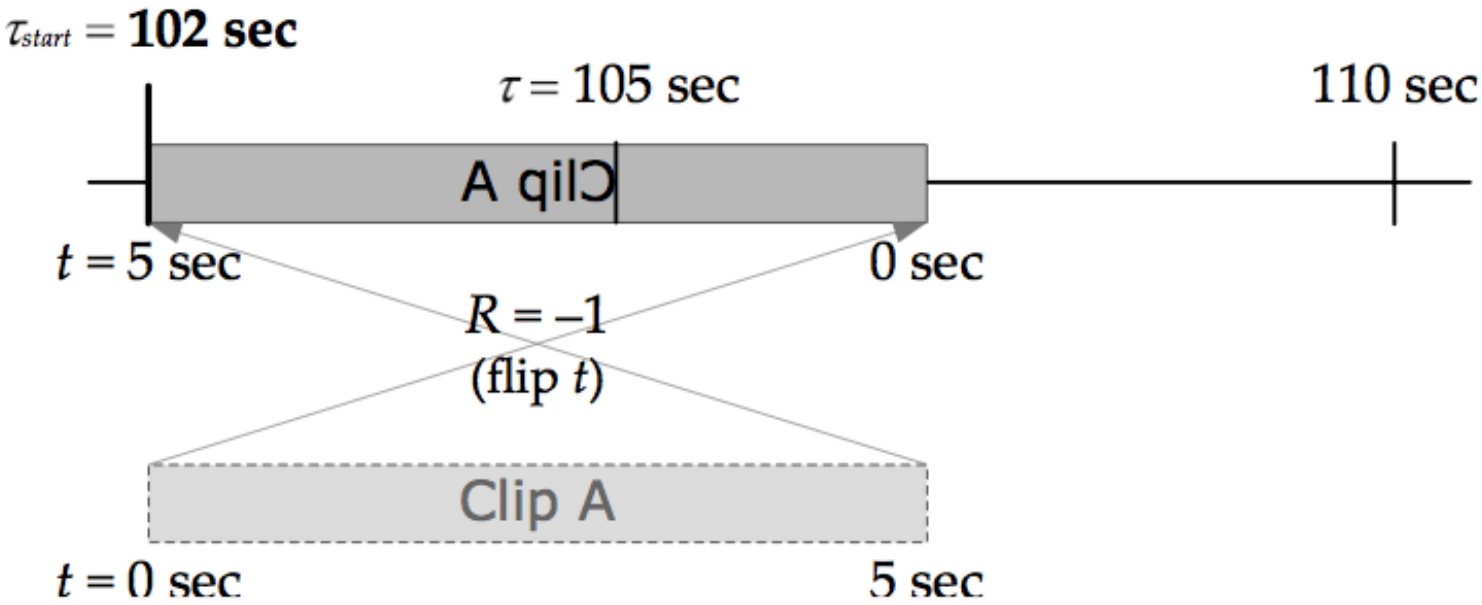
- Real Time
  - measured from CPU clock
- Game Time
  - *usually* coincides with real time
  - can be paused or scaled
- Local Timelines
  - e.g. for animation clips

Wall Clock vs Game Clock

- Rule of Time: Game time is constant regardless of frame rate
  - "Game time" must be aligned to "real time"
    - A car traveling at 60mph must travel the same distance when the game is updating at 60fps at 20 fps.
  - Sometimes "game time" may run at a different speed
    - Sports games sometimes accelerates clock
    - "Game time" doesn't change when paused


τ_{start} = 102 sec
τ = 105 sec
110 sec
Clip A
t = 0 sec
5 sec


τ_{start} = 102 sec
τ = 105 sec
Clip A
t = 0 sec
t = 5 sec
R = 2
(scale t by 1/R = 0.5)
Clip A
t = 0 sec
5 sec


τ_{start} = 102 sec
τ = 105 sec
110 sec
A qilↃ
t = 5 sec
0 sec
R = –1
(flip t)
Clip A
t = 0 sec
5 sec

Subtleties of Time

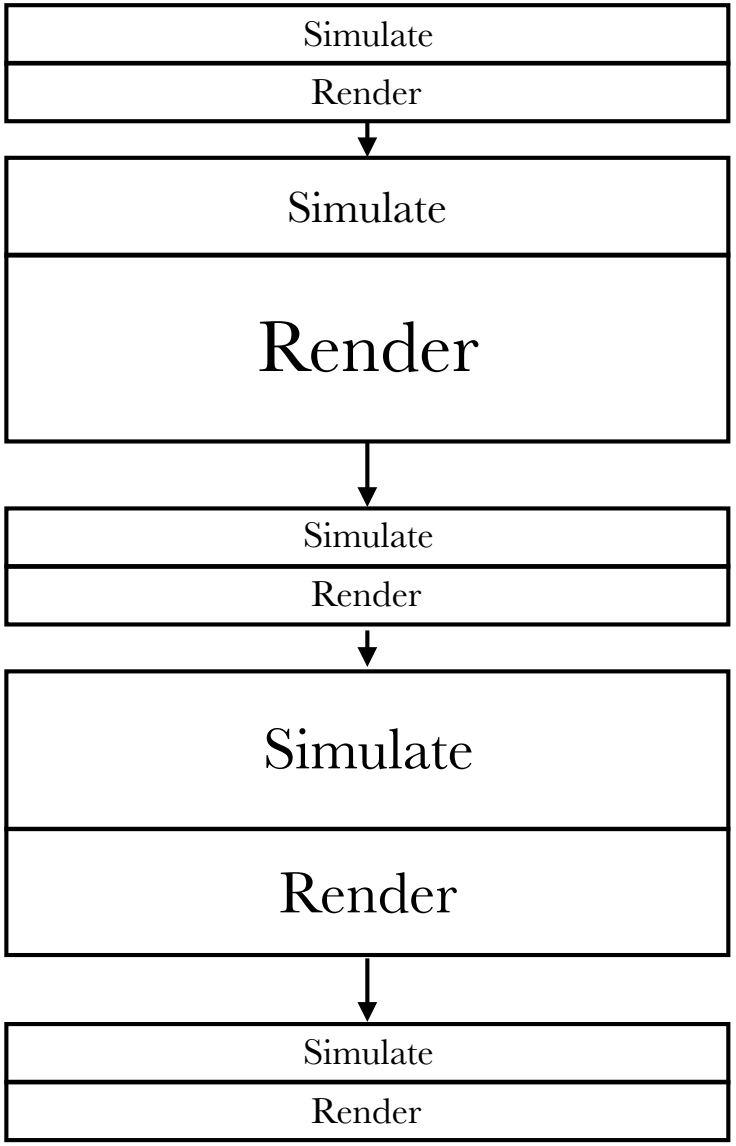
- We typically discern several different clocks
  - Real, Audio, Simulation, Game
- Various game states affect clocks in different ways
  - e.g. pausing may only pause simulation clock
- Must be cautious when interacting systems use different time scales
  - If a cinematic needs to be synced to sound (e.g. lip syncing), may need to drive it directly by the audio clock

Time - Fixed Step

- simplest

```
while(1){
   time += timestep;
   simulate(timestep);
   render()
}
```

Time - Fixed Step



- Advantages:
  - Simple
  - Good for debugging and replay
- Disadvantages:
  - Sensitive to variable work loads
  - One size doesn't fit all
    - Physics needs small Δ
    - AI can go larger

Time - Multiple Simulation Steps

- Fixed time step can be used with multiple simulation steps to deal with slow frame rates

```
while(1){
  now = systemTime();
  while (simetime < now){
    sometime += timestep;
    simulate(timestep);
  }
  render()
}
```

Time - Multiple Simulation Steps

- Advantages
  - Still pretty simple
  - Does well when rendering is slow and simulation is fast
- Disadvantages
  - Everything runs in a multiple of the time step
    - Rendering rate locked to time step
    - Errors must be carefully propagated across updates
    - Possible death spiral – if the sim step is slow, running it multiple times will create more work, which makes the Δ greater, which creates more work…

Time - Variable Step

- another possibility

```
while(1){
   now = SystemTime();
   timestep = now - time;
   time = now;
   simulate(timestep);
   render()
}
```

Time - Variable Step

- Advantages
  - Self-Balancing
  - Works well on platforms where CPU power is variable (PCs)
- Disadvantages
  - Can lead to instability in code that relies on small time step
    - Collision detection/response
  - Non-deterministic
    - Can result in unrepeatable behavior
    - Not as good for instant replay

Time - Subiteration

- Subsystems may “cut up” the variable or fixed step into smaller fixed steps

```
collision_detect(timestamp){
  while (timestep > 0){
    doSomeThings(smallerstep)
    timestep -= smallerstep;
  }
}
```

Time - Subiteration

- Advantages
  - Flexible – systems can operate using a fixed or variable step as needed
  - Sensitive systems can use a smaller step
- Disadvantages
  - Increased complexity
  - Systems operating at different update rates may experience undesired interactions
    - Temporal aliasing (strobing)

From Frame Rate to Speed

$$\Delta x = v\Delta t$$

$$x_2 = x_1 + \Delta x = x_1 + v\Delta t$$

- Explicit Euler numerical integration
- only works well with constant speeds
- there's more complex numerical integration with dynamics, but all techniques make use of elapsed frame time in some way

Time

- Inconsistent handling of time throughout the game will create difficult to fix bugs
- Be clear as to the units of time used in variable names and function calls

```
void Update( float time ) { ... }  // Bad

void Update( float milliseconds ) { ... } // Still bad

void UpdateMilliseconds( float time ) { ... }  // Good

void Update( Time& time ) { ... }  // Good
```

Updating Based on Elapsed Time

- Measure time at beginning and end of every frame, send difference to relevant subsystems
- Susceptible to frame-rate spikes
  - Can take running average
  - Can govern (lock) frame rate
    - good for physics
    - can make record and playback more reliable

High-Resolution Timers

- `time()`
  - number of seconds since 1/1/1970
  - bad!
- `rtdsc` (Pentium)
- `QueryPerformanceCounter()` (Win32)
- `mftb` (XBox360 and PS3)
- `mfspr` (other PowerPC)

64-bit Register

- Most processor use this
- At 3GHz, will wrap approximately every 195 years
- If 32-bit, would wrap every 1.4 seconds
- Choose data types wisely!

Subtract, then Convert

- `tBegin = 0x12345678FFFFFFB7`
- `tEnd   = 0x1234567900000039`

Break Points

```
//Assume “ideal” dt (30FPS)
F32 dt = 1.0f / 32.0f;
//Read current time
U64 tBegin = readHiResTimer()

while (true) //main loop
{
  updateSubsystemA(dt);
  updateSubsystemB(dt);
  //etc...
  renderScene();
  swapBuffers();

  U64 tEnd = readHiResTimer();
  dt = (F32)(tEnd - tBegin)

  //frame lock if dt is too large
  if (dt > 1.0f/10.0f)
  {
    dt = 1.0f / 30.0f
  }
  tBegin = tEnd;
}
```

Clock Class

```
class Clock
{
  U64  m_timeCycles;
  F32  m_timeScale;
  bool m_isPaused;

  static F32 s_cyclesPerSecond;

  static inline U64 secondsToCycles(F32 timeSeconds)
  {
    return (U64)(timeSeconds * s_cyclesPerSecond);
  }

  //only for small durations!
  static inline F32 cyclesToSeconds(U64 timeCycles)
  {
    return (F32)timeCycles / s_cyclesPerSecond;
  }
```

Clock Class

```
public:
  static void init() //call when game first starts up
  {
    s_cyclesPerSecond = (F32)readHiResTimerFrequency();
  }

  explicit Clock(F32 startTimeSeconds = 0.0f):
    m_timeCycles(secondsToCycles(startTimeSeconds)),
    m_timeScale(1.0f), //default to unscaled
    m_isPaused(false)  //default to running
  {
  }

  F32 calcDeltaSeconds(const Clock& other)
  {
    U64 dt = m_timeCycles - other.m_timeCycles;
    return cyclesToSeconds(dt);
  }
```

Clock Class

```
void update(F32 dtRealSeconds)
{
  if (!m_isPaused)
  {
    U64 dtScaledCycles = secondsToCycles(
      dtRealSeconds * m_timeScale);
    m_timeCycles += dtScaledCycles;
  }
}

void singleStep() //just add one frame interval
{
  if (m_isPaused)
  {
    U64 dtScaledCycles = secondsToCycles(
      (1.0f / 30.0f) * m_timeScale);
    m_timeCycles += dtScaledCycles;
  }
}
```