Developing a Multiplayer RTS with the Unreal Engine 3

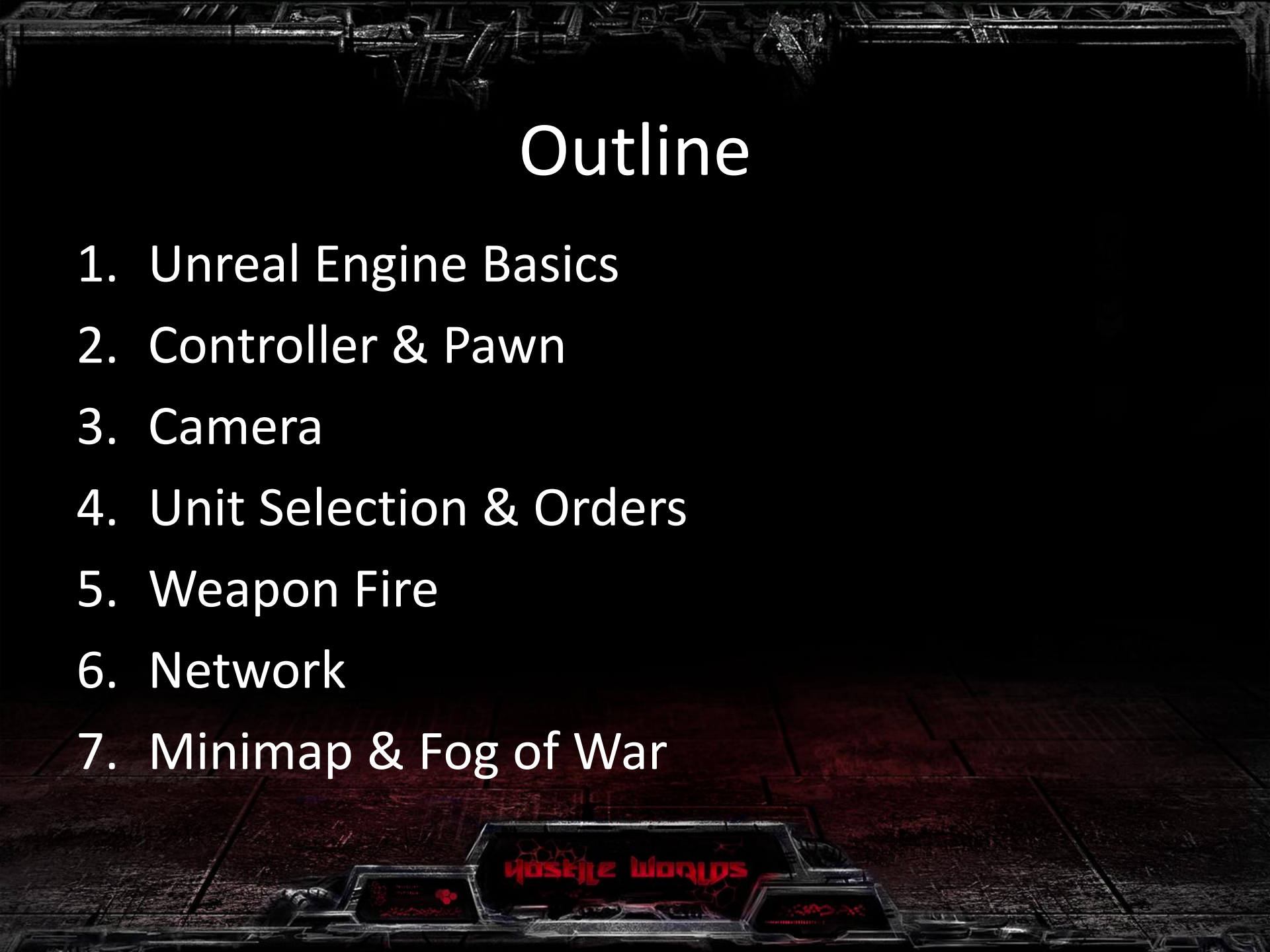
Marcel Köhler, Nick Prühs

Faculty of Design, Media and Information
Hamburg University of Applied Sciences

January 17, 2011

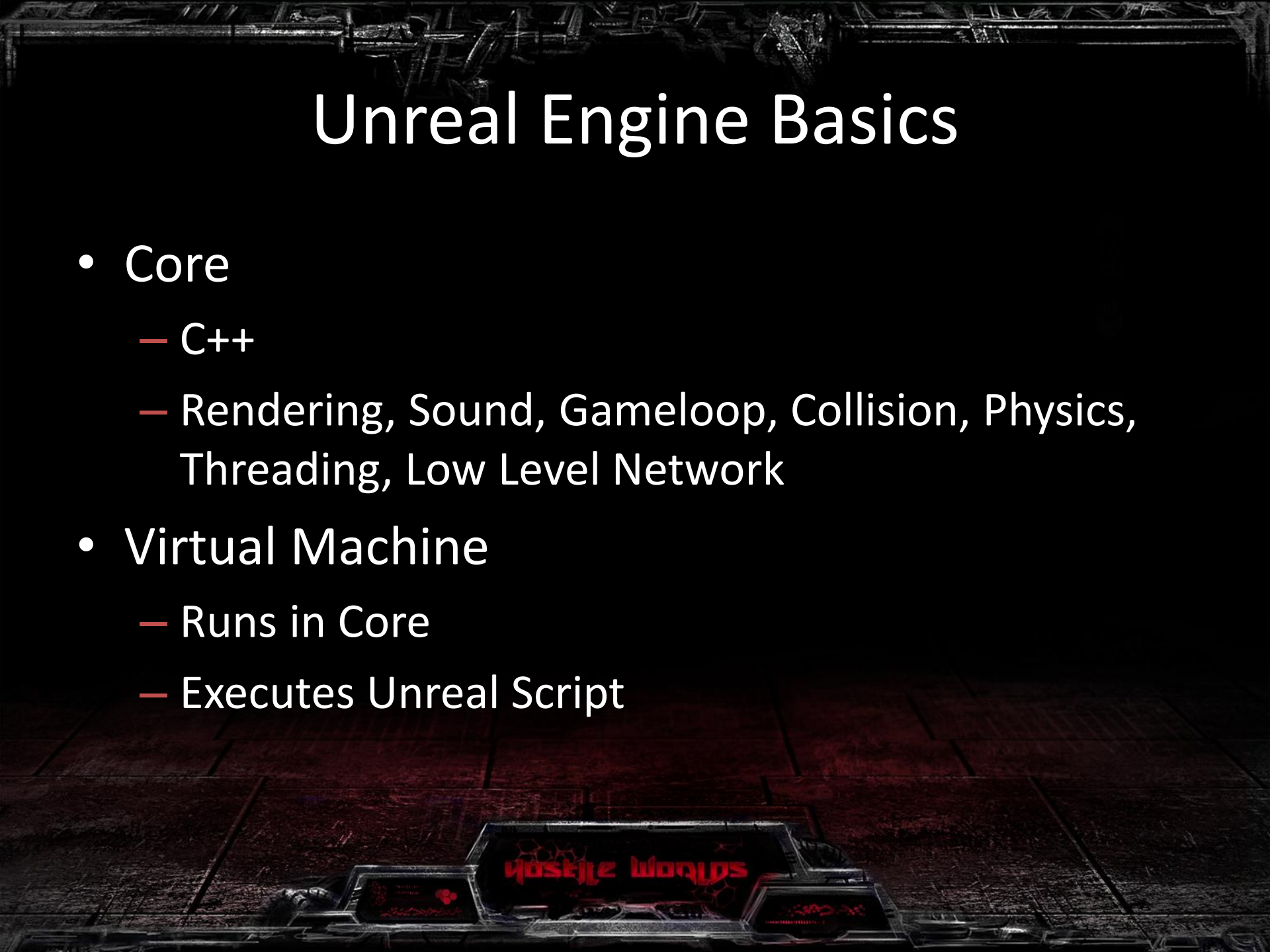
Outline

1. Unreal Engine Basics
2. Controller & Pawn
3. Camera
4. Unit Selection & Orders
5. Weapon Fire
6. Network
7. Minimap & Fog of War



Unreal Engine Basics

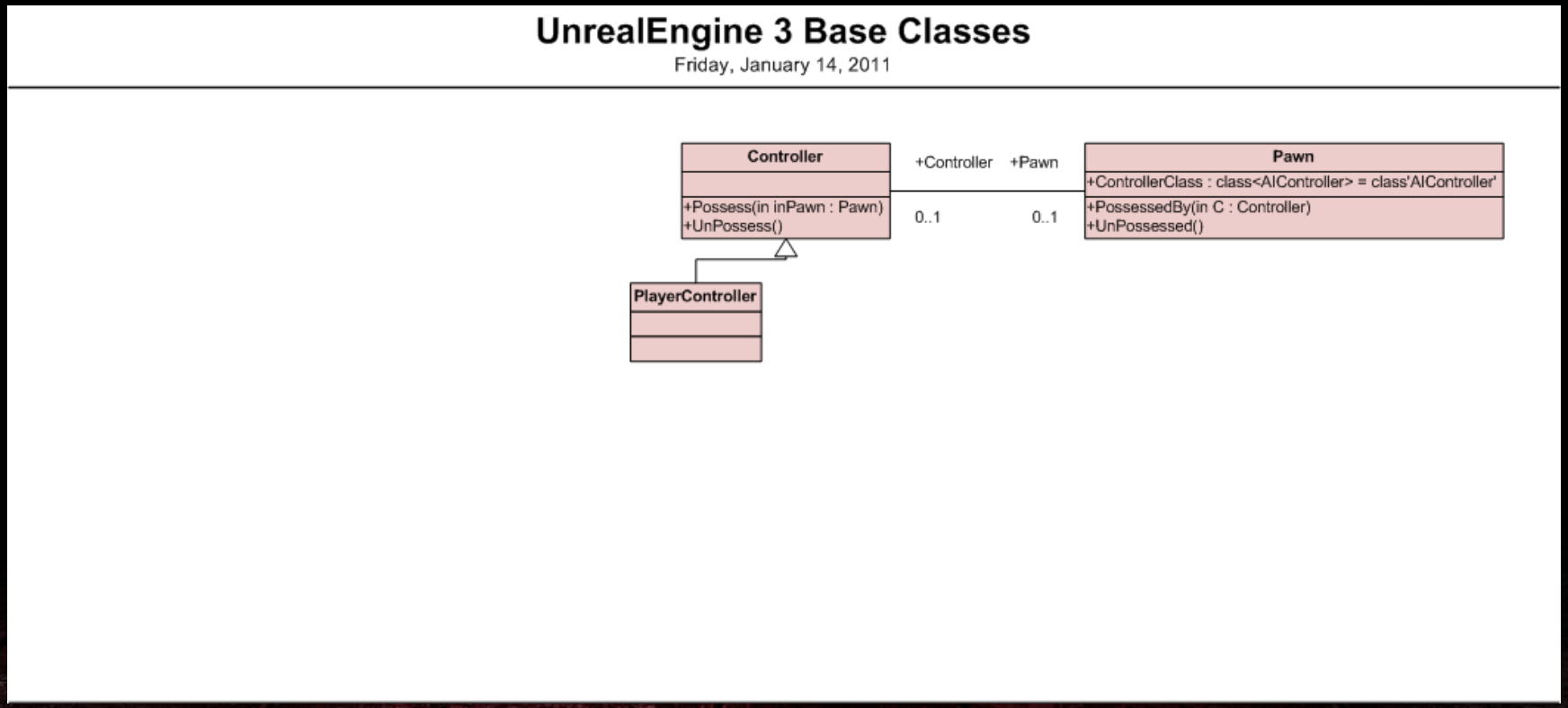
- Core
  - C++
  - Rendering, Sound, Gameloop, Collision, Physics, Threading, Low Level Network
- Virtual Machine
  - Runs in Core
  - Executes Unreal Script



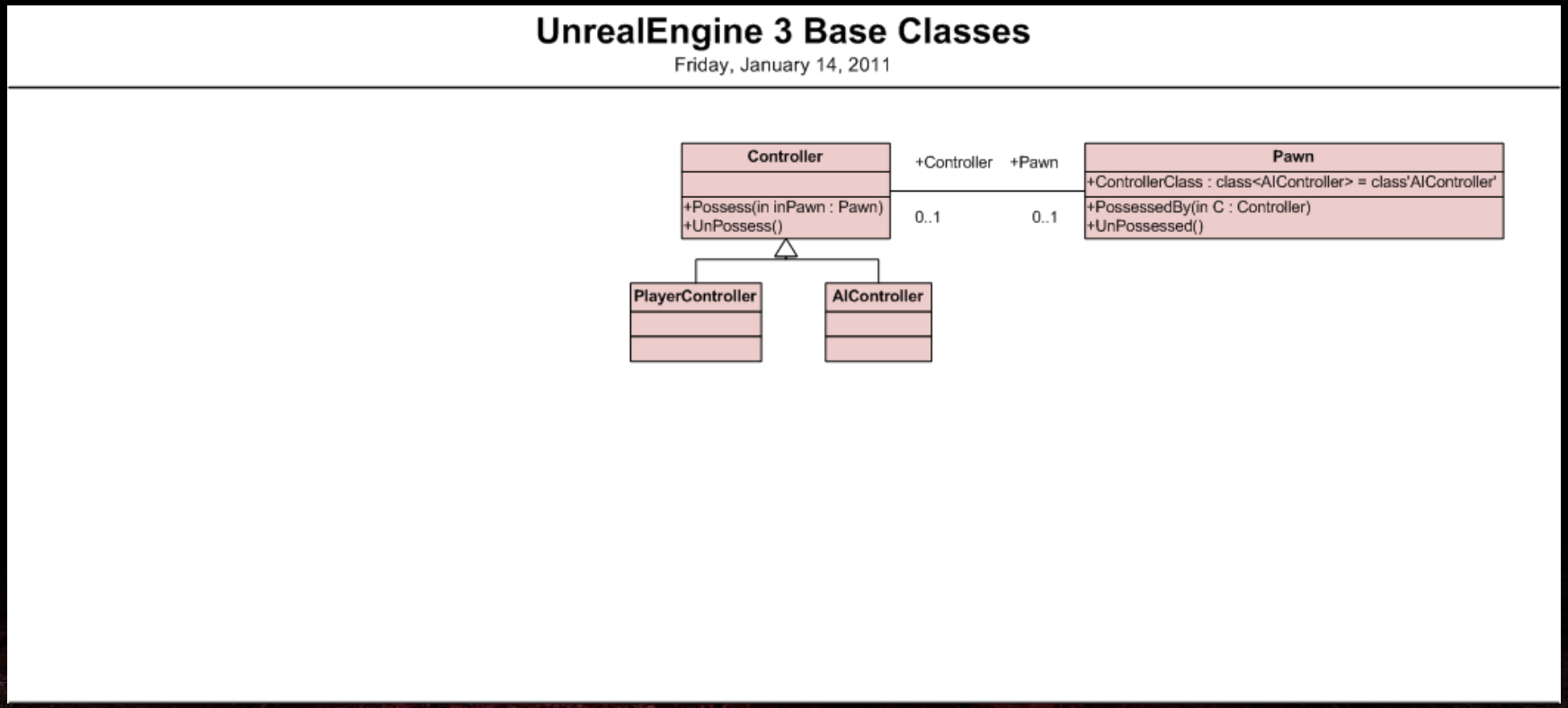
Unreal Engine Basics

- Unreal Script
  - Similar to C++ and Java
  - High-level object-oriented language
  - Bytecode based (platform independent)
  - Pointerless environment with automatic garbage collection
  - Simple single-inheritance class graph
  - Strong compile-time type checking
  - Safe client-side execution "sandbox"

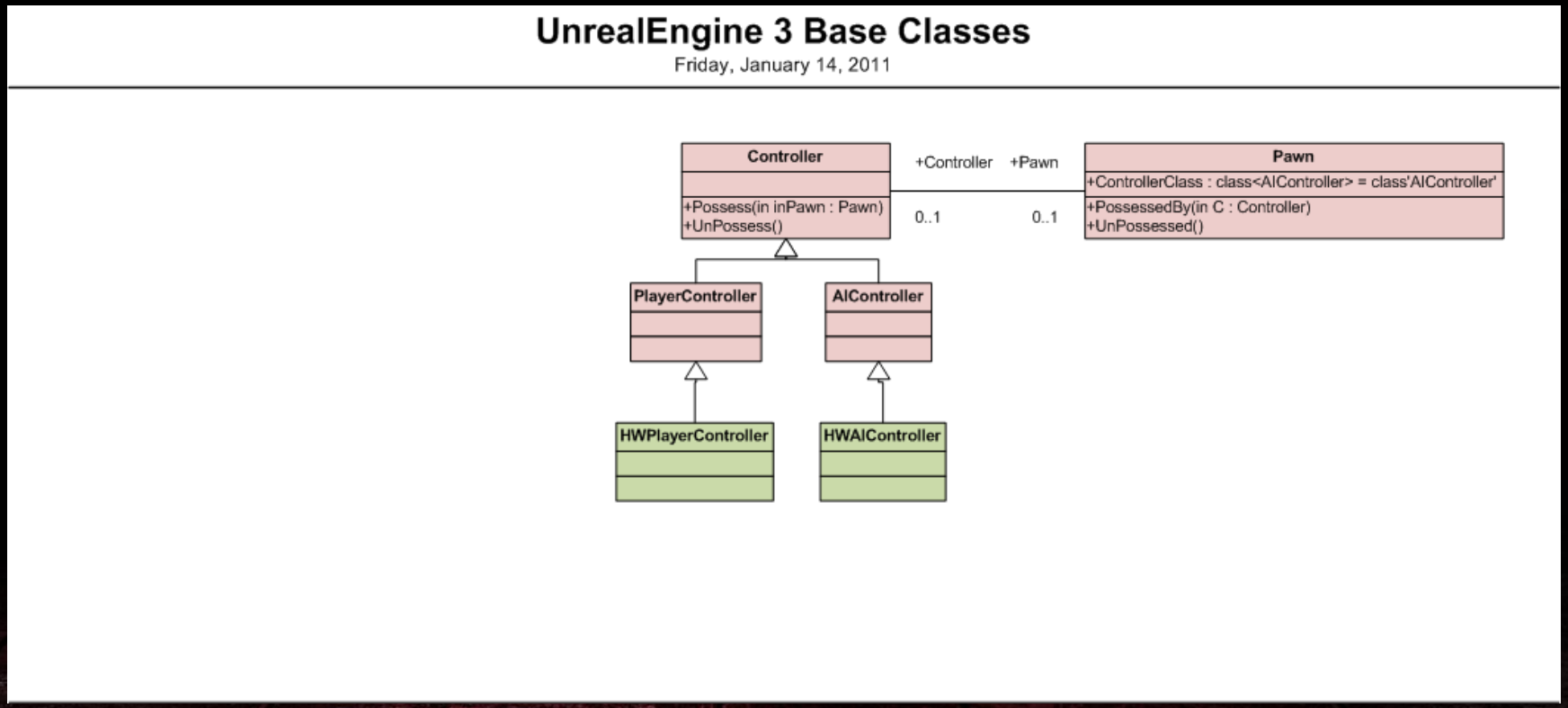
Controller & Pawn



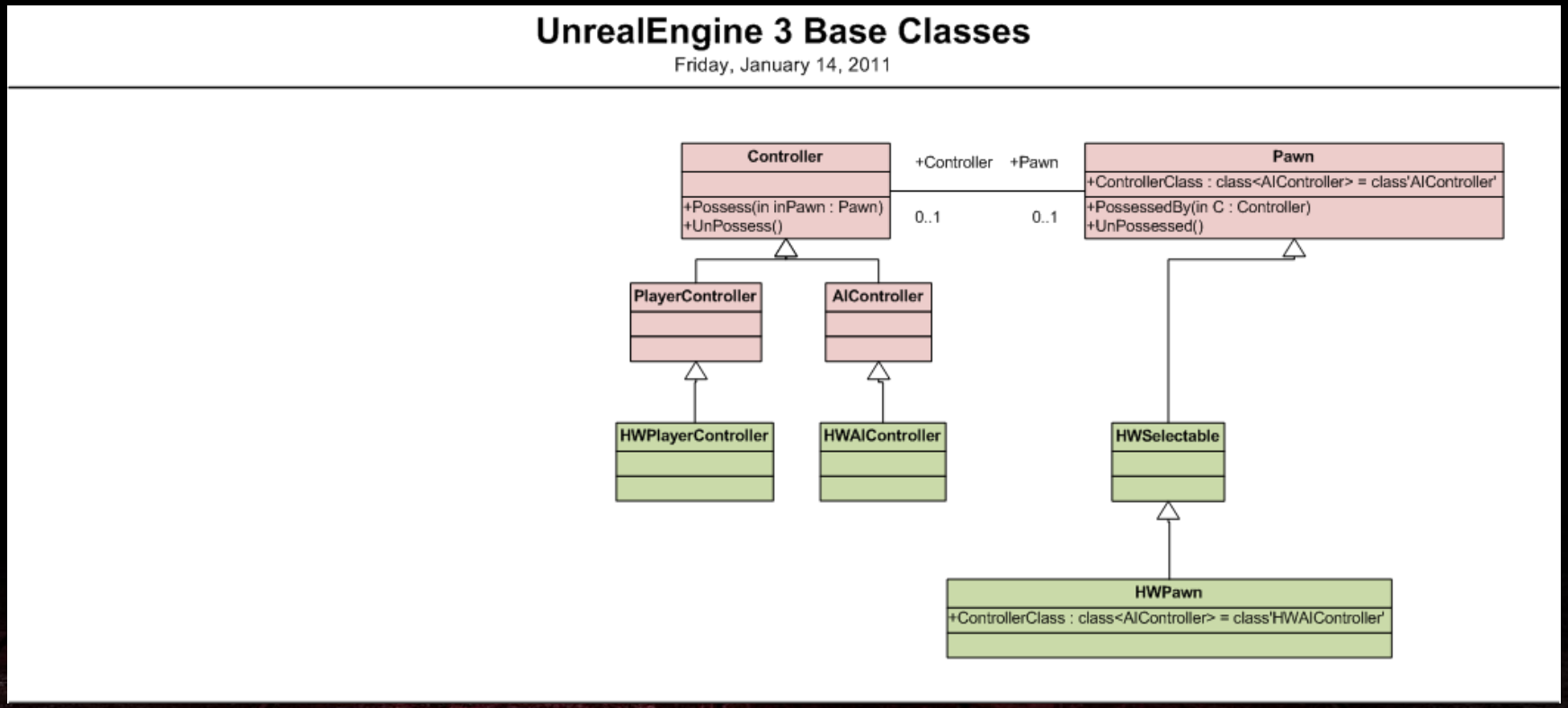
Controller & Pawn



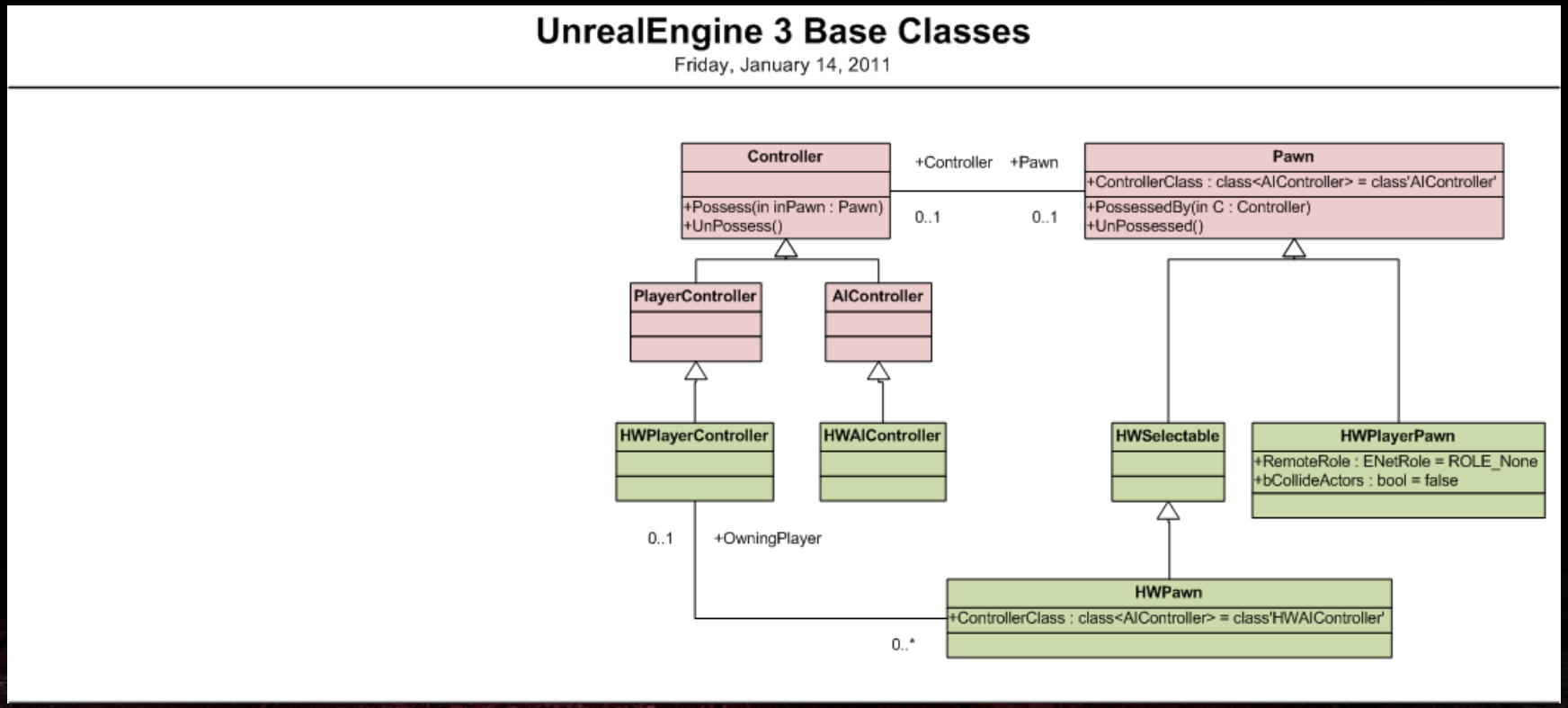
Controller & Pawn



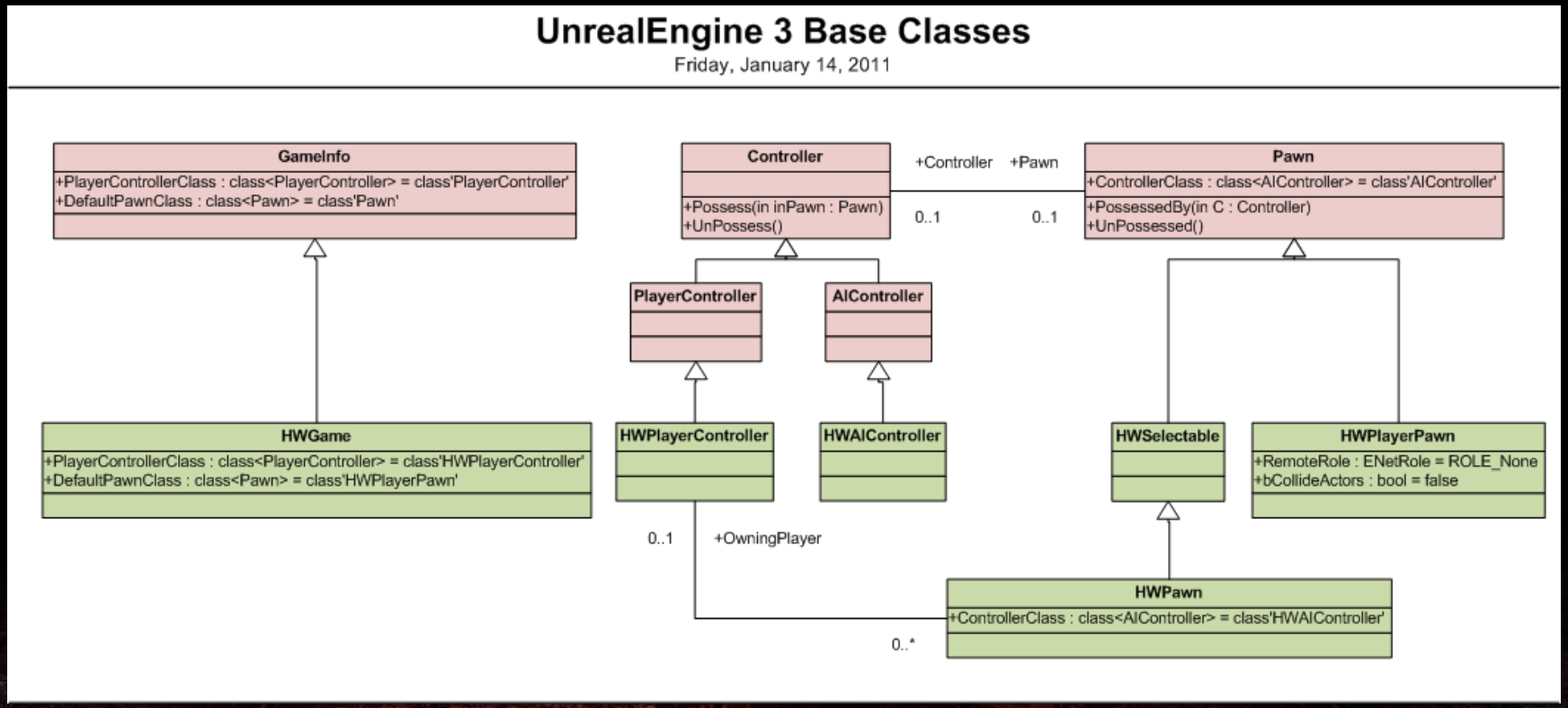
Controller & Pawn



Controller & Pawn



Controller & Pawn



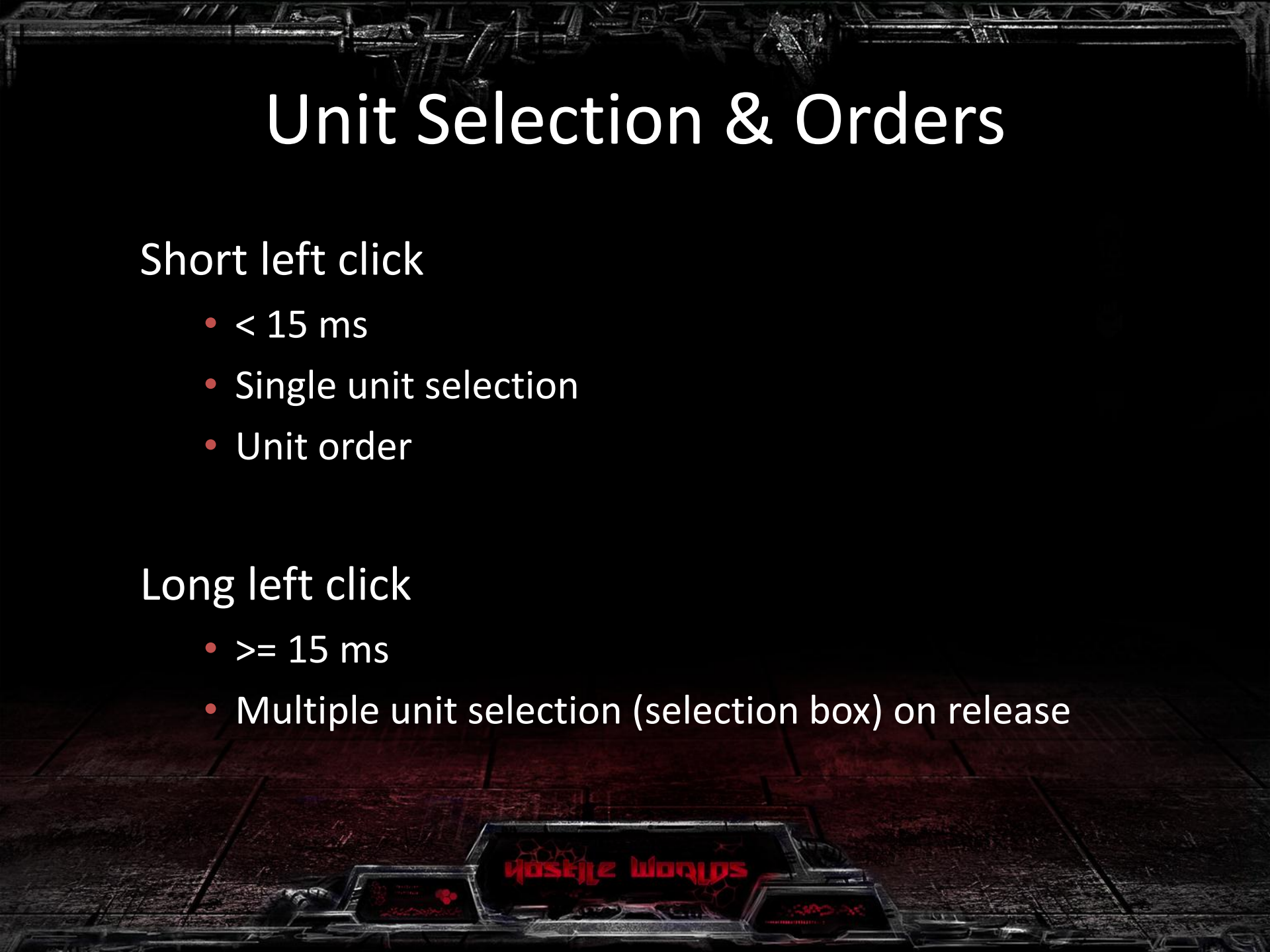
Unit Selection & Orders

Short left click

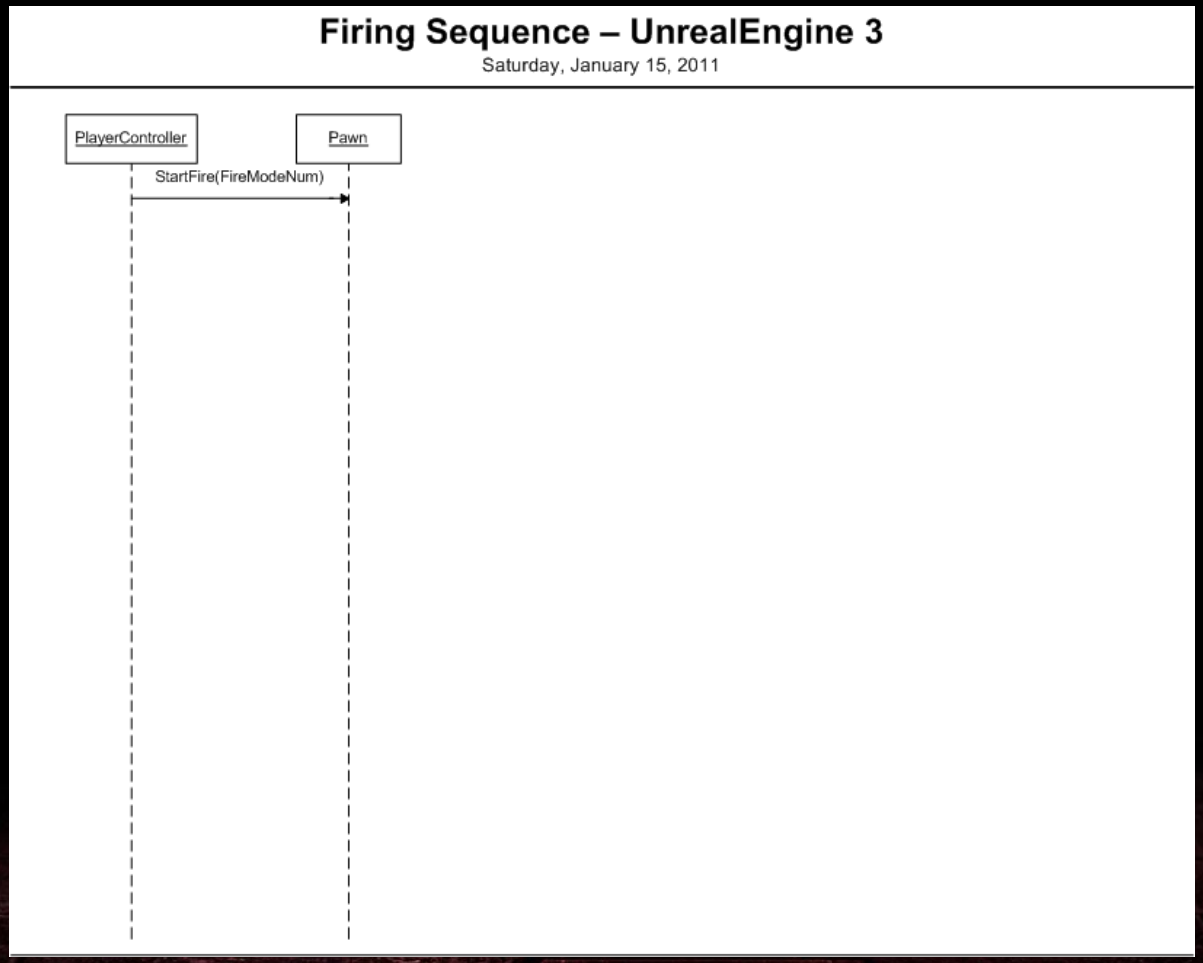
- < 15 ms
- Single unit selection
- Unit order

Long left click

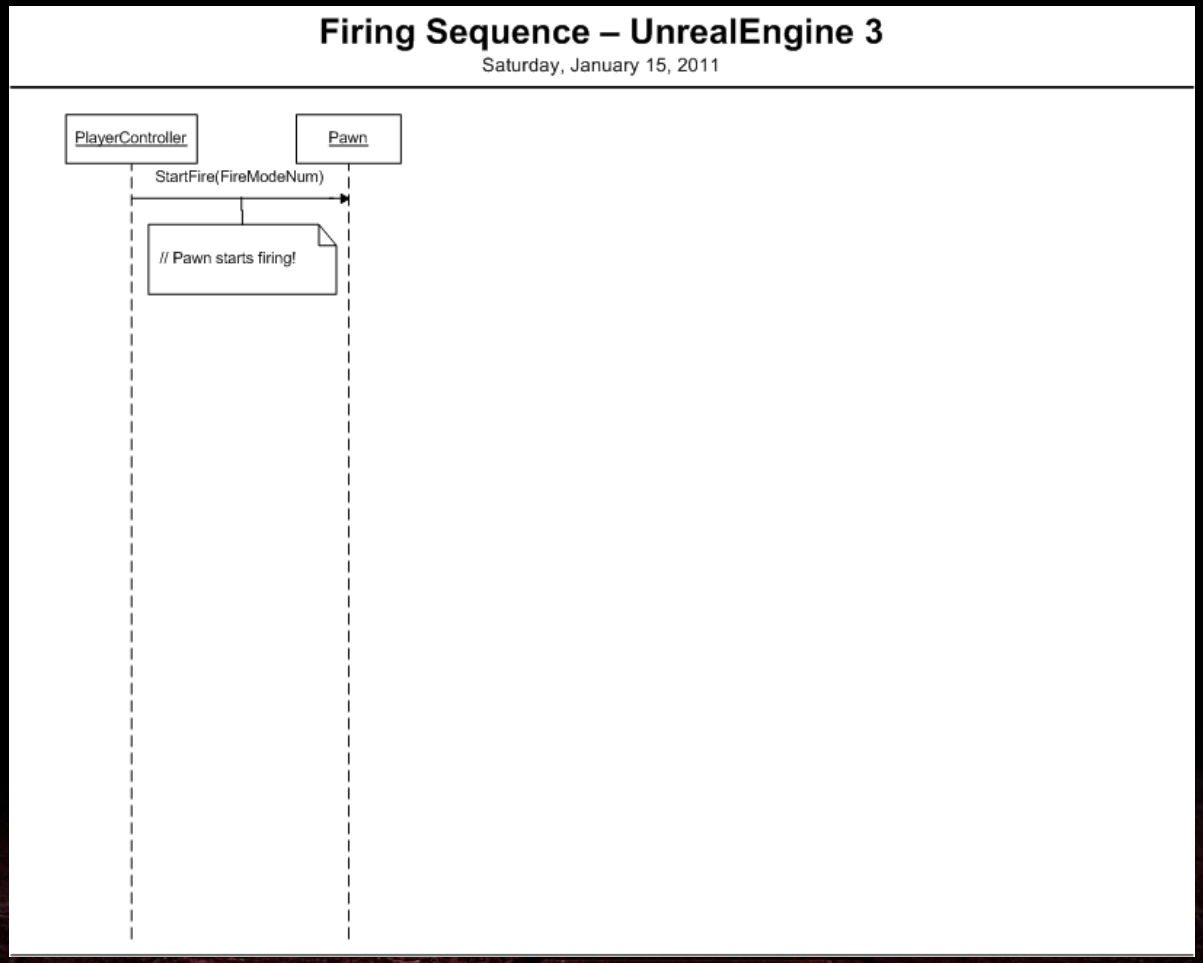
- >= 15 ms
- Multiple unit selection (selection box) on release



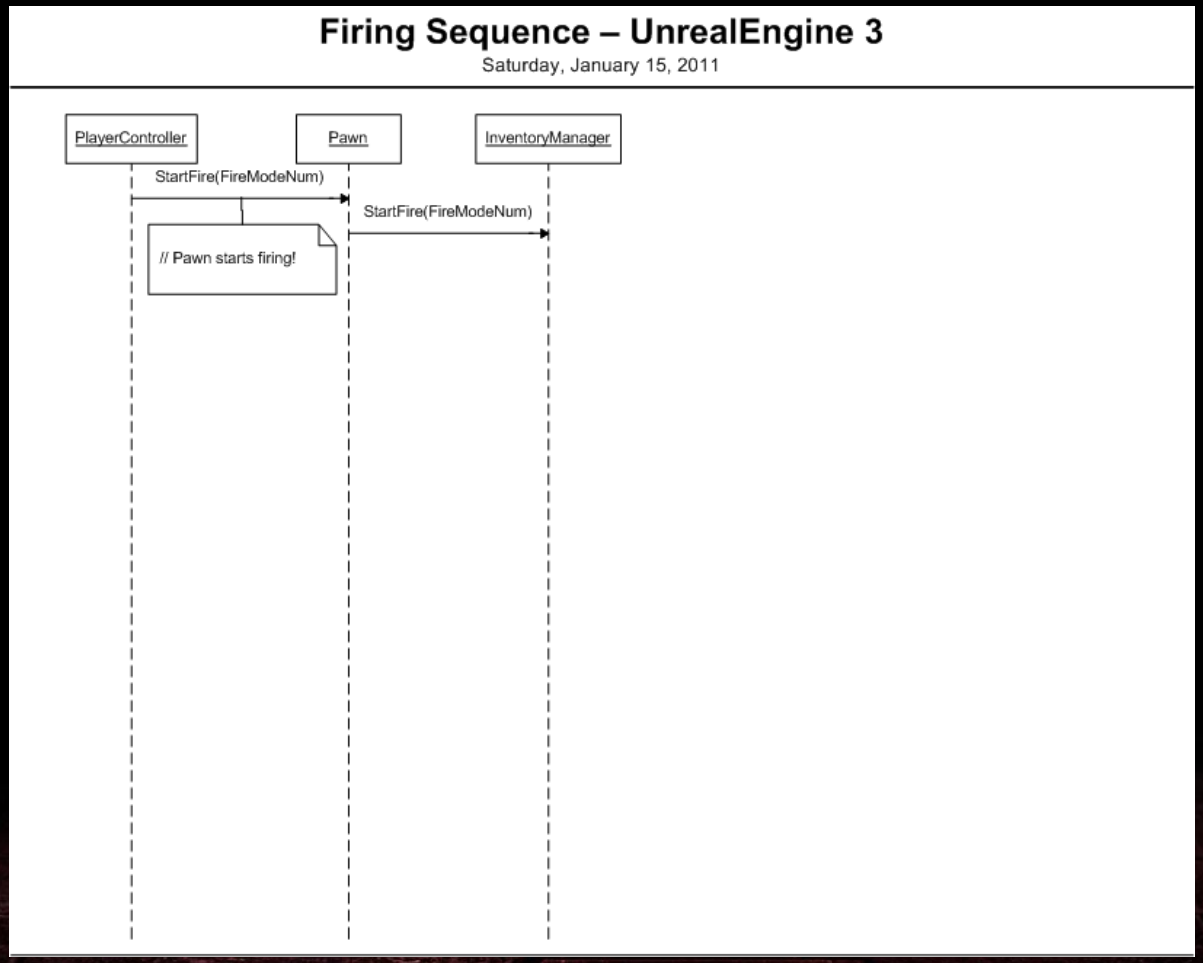
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
StartFire(FireModeNum)

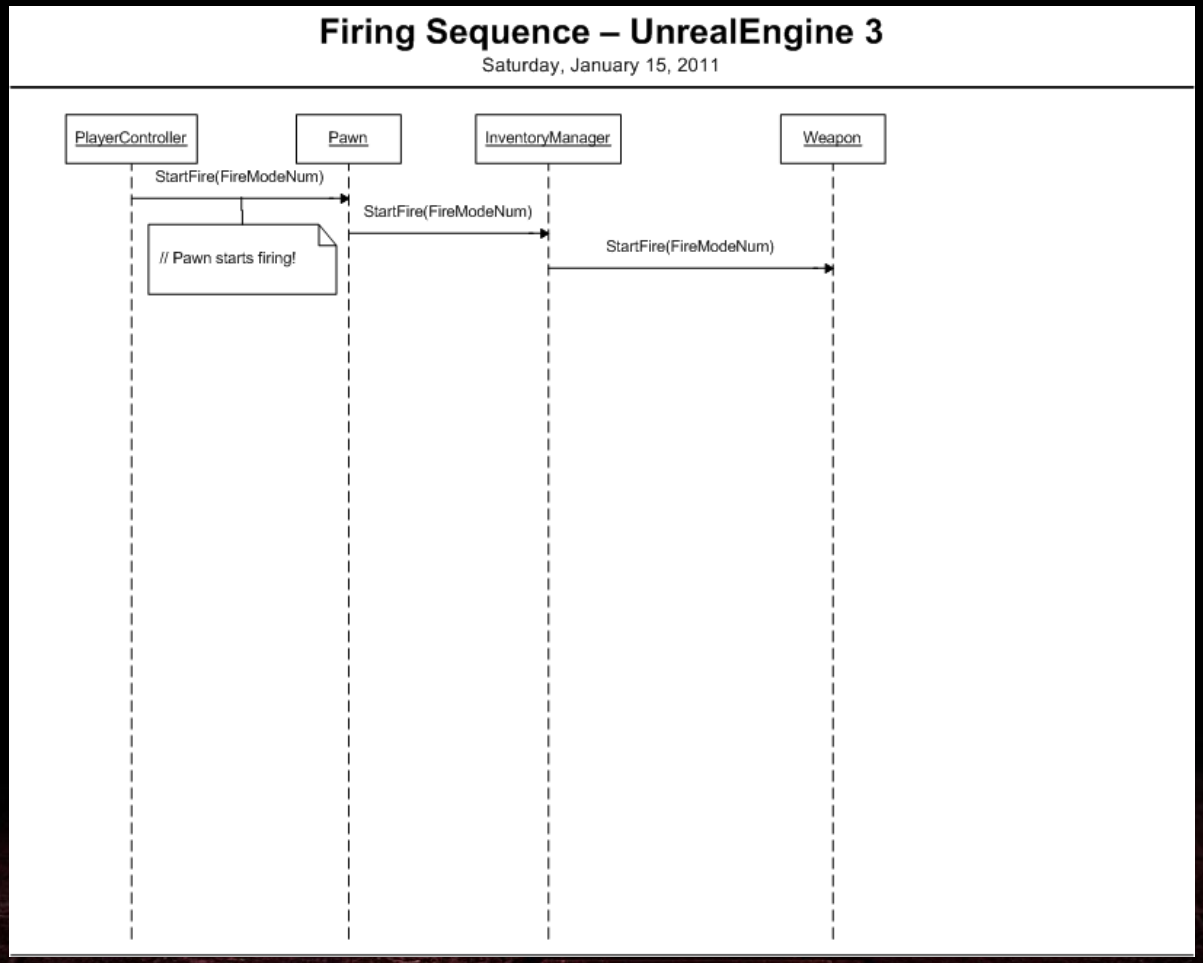
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
StartFire(FireModeNum)
// Pawn starts firing!

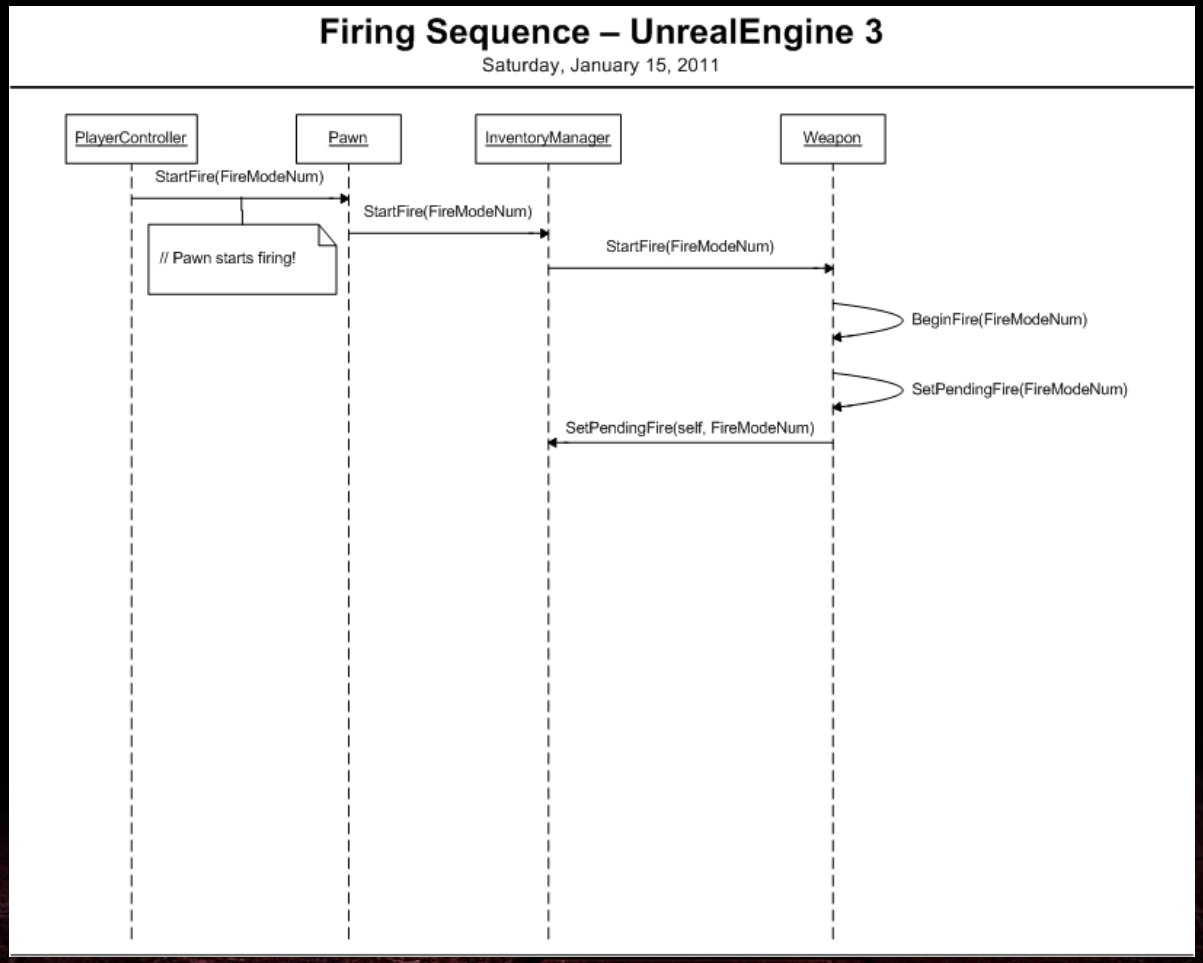
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!

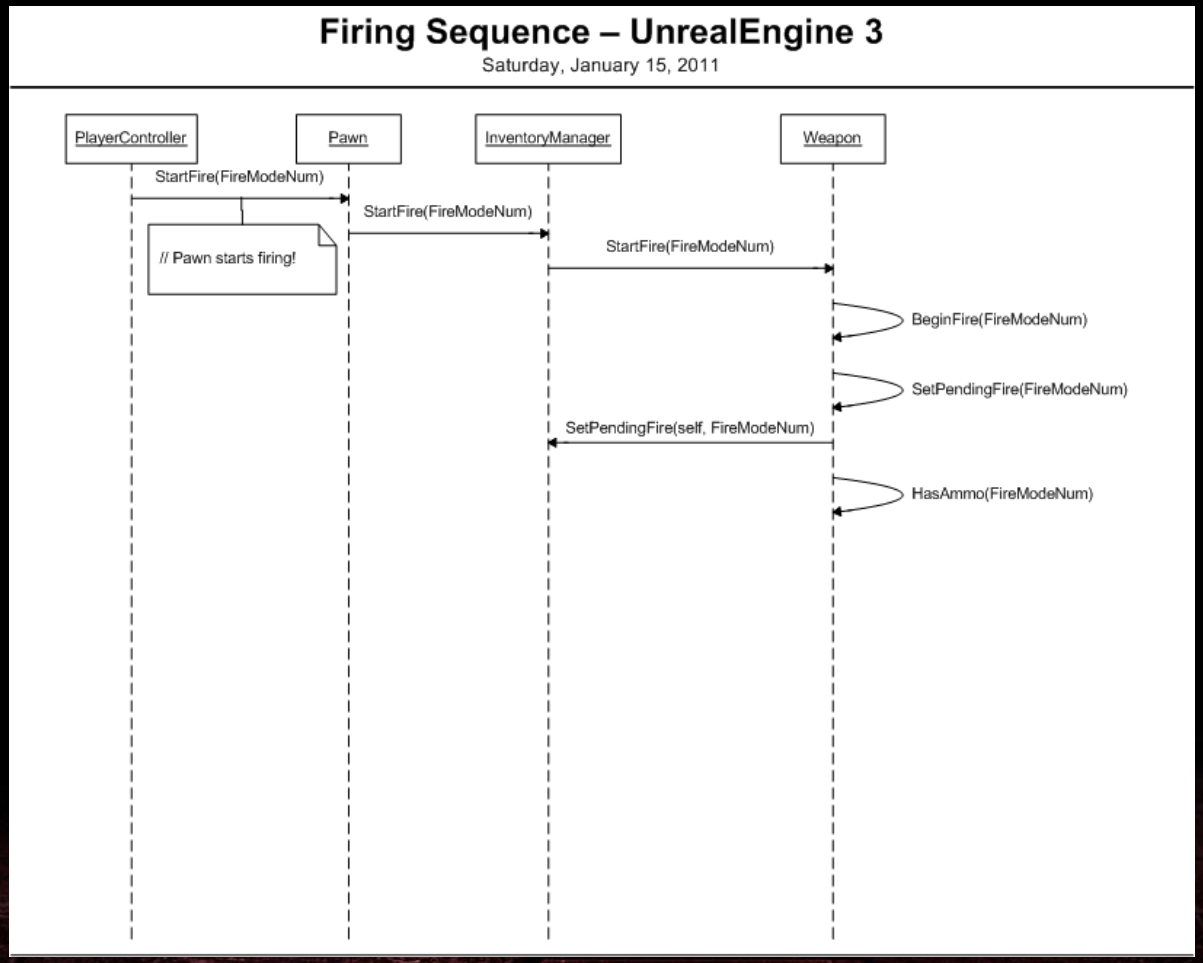
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
Weapon
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!
StartFire(FireModeNum)

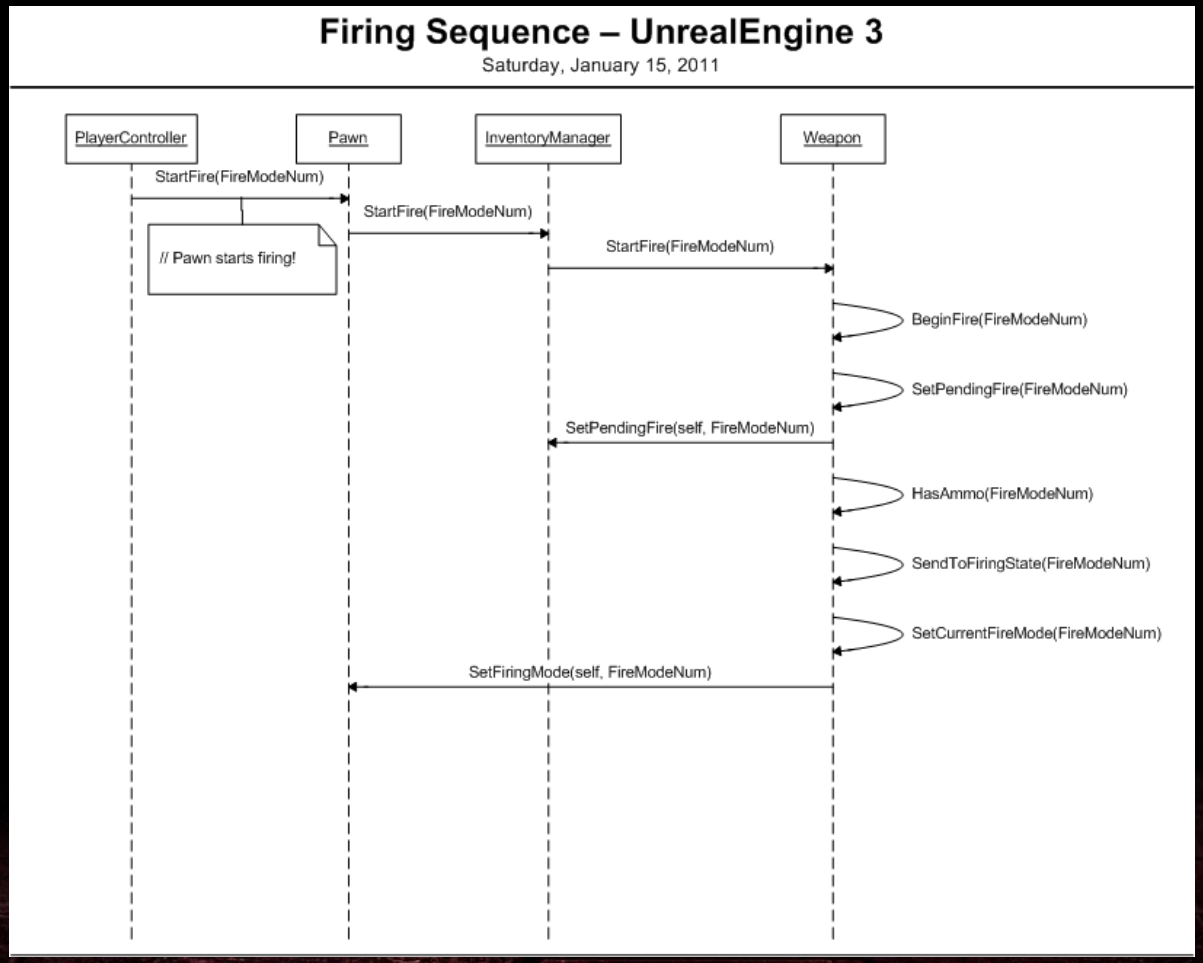
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
Weapon
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!
StartFire(FireModeNum)
BeginFire(FireModeNum)
SetPendingFire(FireModeNum)
SetPendingFire(self, FireModeNum)

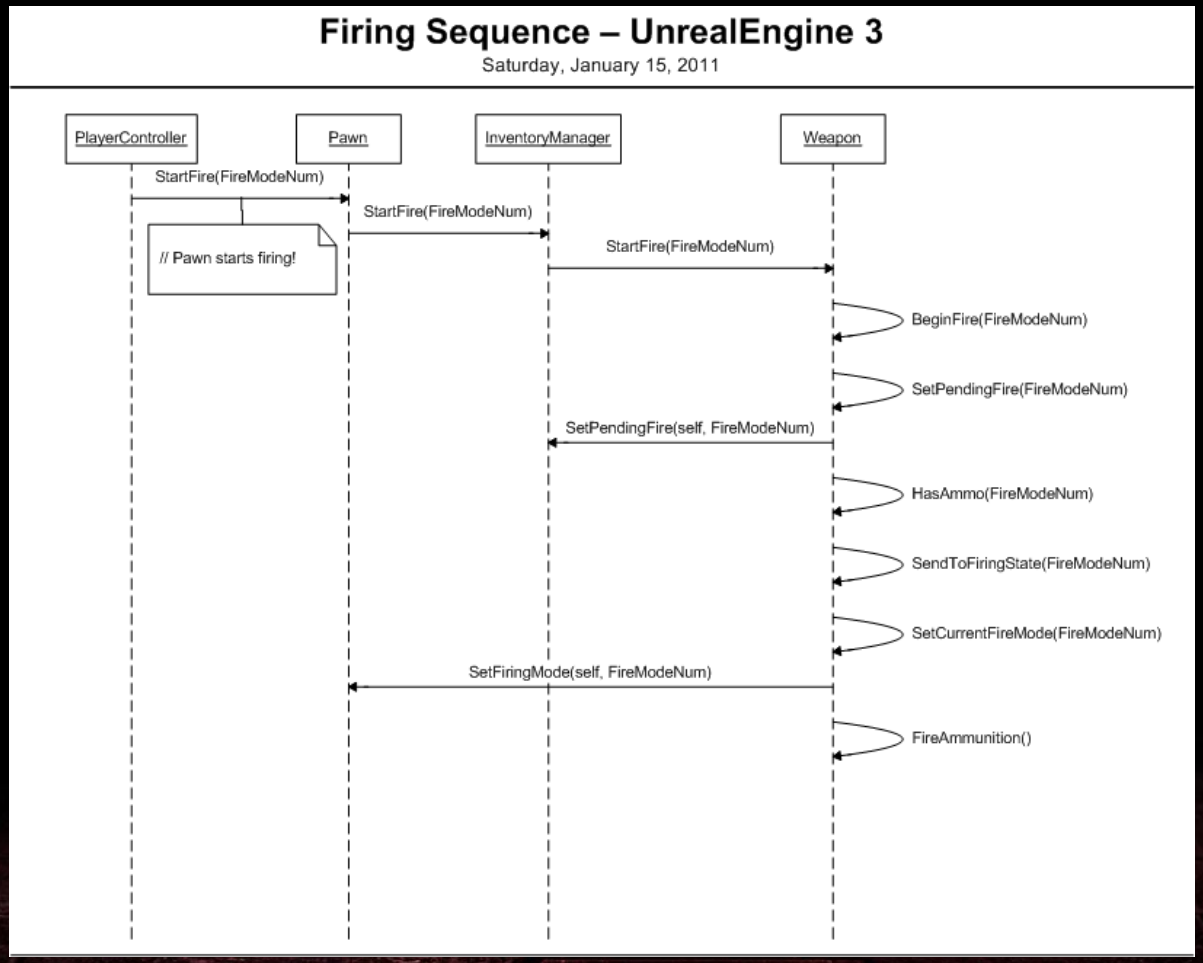
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
Weapon
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!
StartFire(FireModeNum)
BeginFire(FireModeNum)
SetPendingFire(FireModeNum)
SetPendingFire(self, FireModeNum)
HasAmmo(FireModeNum)

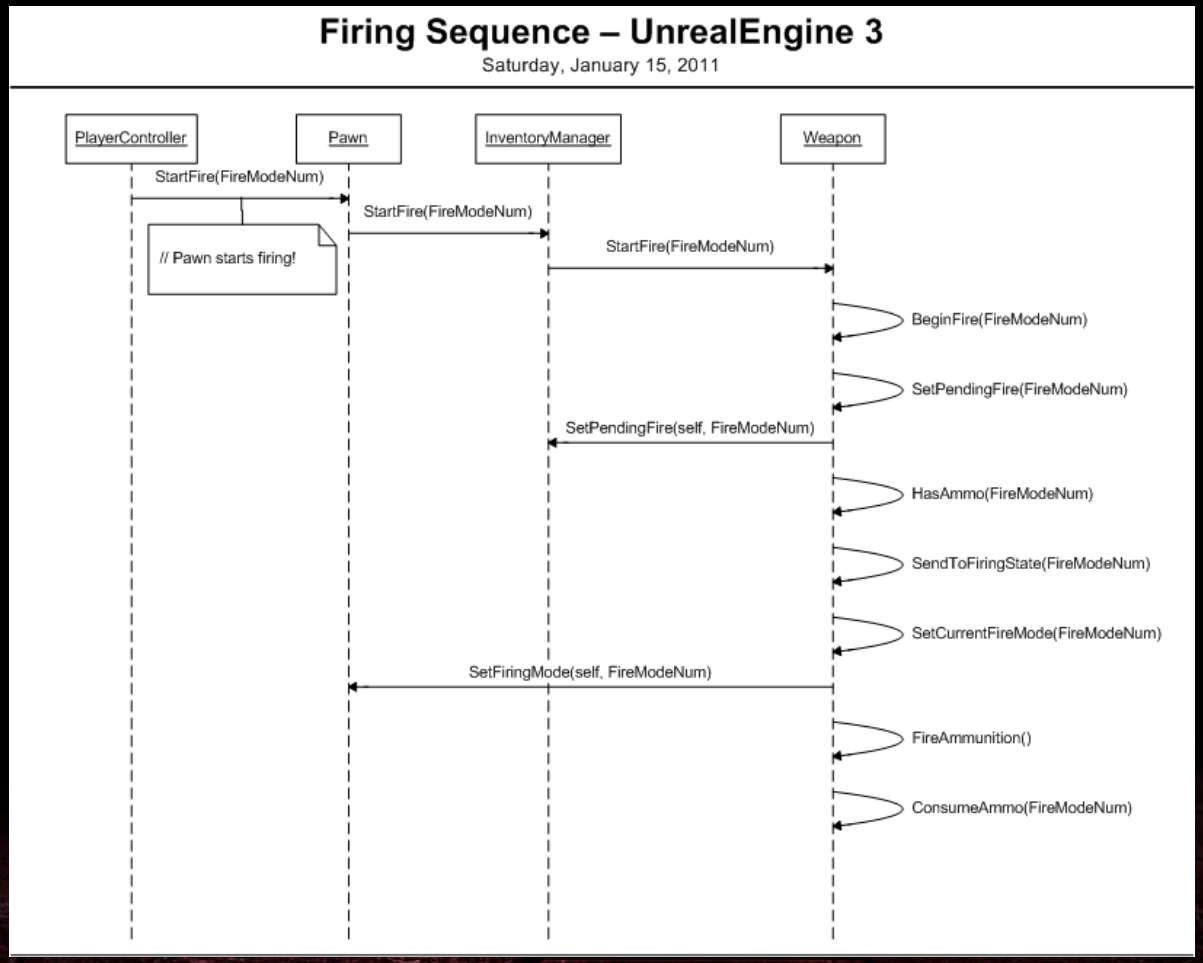
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
Weapon
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!
StartFire(FireModeNum)
BeginFire(FireModeNum)
SetPendingFire(FireModeNum)
SetPendingFire(self, FireModeNum)
HasAmmo(FireModeNum)
SendToFiringState(FireModeNum)
SetCurrentFireMode(FireModeNum)
SetFiringMode(self, FireModeNum)

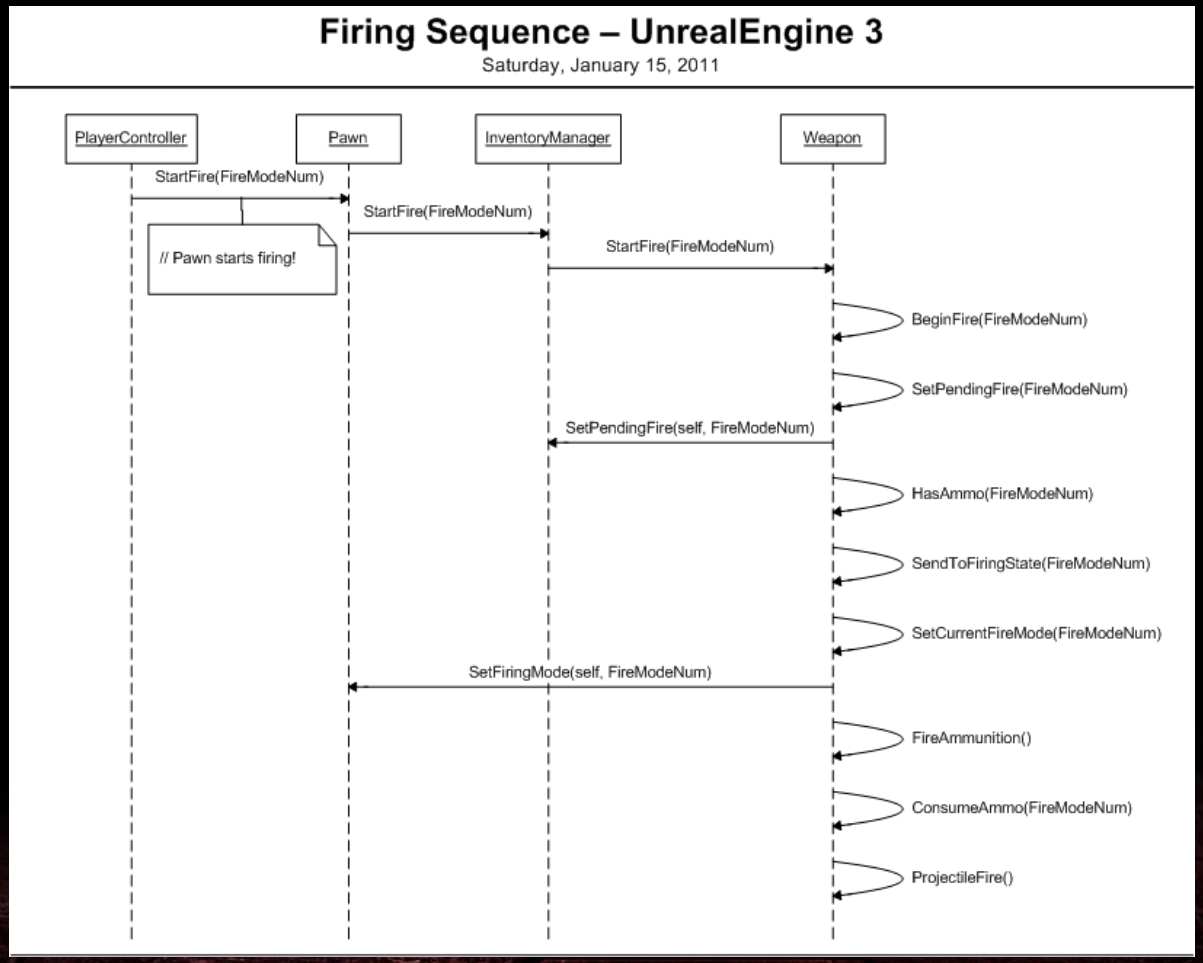
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
Weapon
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!
StartFire(FireModeNum)
BeginFire(FireModeNum)
SetPendingFire(FireModeNum)
SetPendingFire(self, FireModeNum)
HasAmmo(FireModeNum)
SendToFiringState(FireModeNum)
SetCurrentFireMode(FireModeNum)
SetFiringMode(self, FireModeNum)
FireAmmunition()

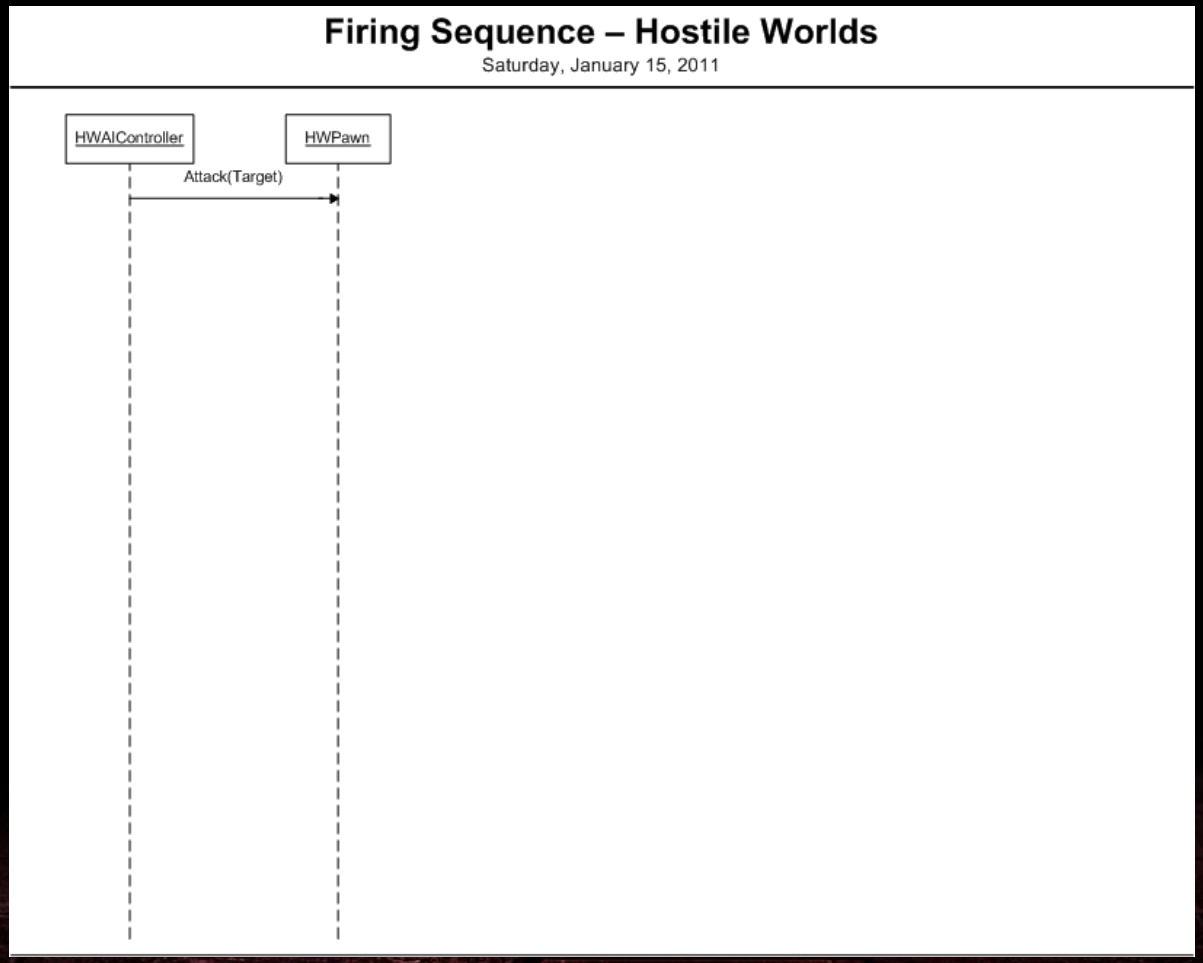
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
Weapon
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!
StartFire(FireModeNum)
BeginFire(FireModeNum)
SetPendingFire(FireModeNum)
SetPendingFire(self, FireModeNum)
HasAmmo(FireModeNum)
SendToFiringState(FireModeNum)
SetCurrentFireMode(FireModeNum)
SetFiringMode(self, FireModeNum)
FireAmmunition()
ConsumeAmmo(FireModeNum)

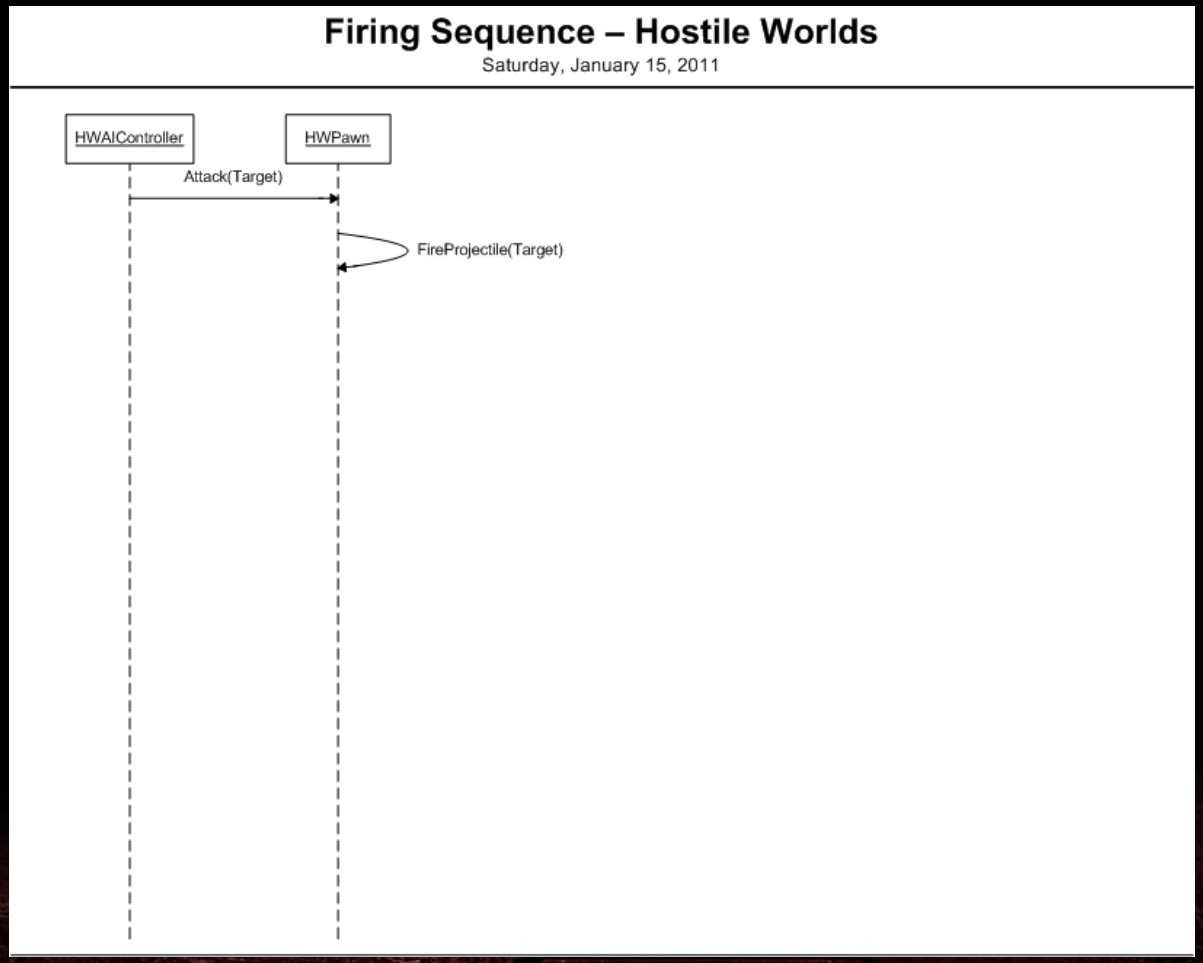
Weapon Fire


Firing Sequence – UnrealEngine 3
Saturday, January 15, 2011
PlayerController
Pawn
InventoryManager
Weapon
StartFire(FireModeNum)
StartFire(FireModeNum)
// Pawn starts firing!
StartFire(FireModeNum)
BeginFire(FireModeNum)
SetPendingFire(FireModeNum)
SetPendingFire(self, FireModeNum)
HasAmmo(FireModeNum)
SendToFiringState(FireModeNum)
SetCurrentFireMode(FireModeNum)
SetFiringMode(self, FireModeNum)
FireAmmunition()
ConsumeAmmo(FireModeNum)
ProjectileFire()

Weapon Fire


Firing Sequence – Hostile Worlds
Saturday, January 15, 2011
HWAIController
HWPawn
Attack(Target)

Weapon Fire


Firing Sequence – Hostile Worlds
Saturday, January 15, 2011
HWAIController
HWPawn
Attack(Target)
FireProjectile(Target)

Network

„Unreal views the general problem of *coordinating a reasonable approximation of a shared reality between the server and clients* as a problem of *replication*.
That is, a problem of determining a set of data and commands that flow between the client and server in order to achieve that approximate shared reality. “

- Tim Sweeney, Epic Games Inc.

Network

- Generalized Client-Server Model
  - Authoritative server (Dedicated, Listen)
  - Predicting and simulating clients
  - Decoupling of Network and Gamelogic facilitates extensibility
    - The Network code can coordinate any game which can be described by the language
    - Network is controlled on language level through keywords & variables
    - Low level network (Serialization, Reliable UDP) done by Core
  - “Hybrid” Code
    - Client and Server execute same code on approximately the same data -> minimizes traffic

Network - Basic Terminology

- Actor
  - Object that can move and interact with other actors
- Level
  - Object which contains a set of actors
- Game State
  - The complete set of all actors that exist in a level
  - The current values of all actor variables

Network – Update Loop

1. if (server)
   Send(Gamestate) to all clients
2. if (client)
   Send(RequestedMovement) to server
   Receive(Gamestate) from server
   Render(ApproximateWorldView) to screen
3. if (server || client)
   Tick(DeltaTime) to update Gamestate
   Update(Actors)
   Execute(Physics)
   Receive(GameEvents)
   Execute(ScriptCode)

Actor Roles

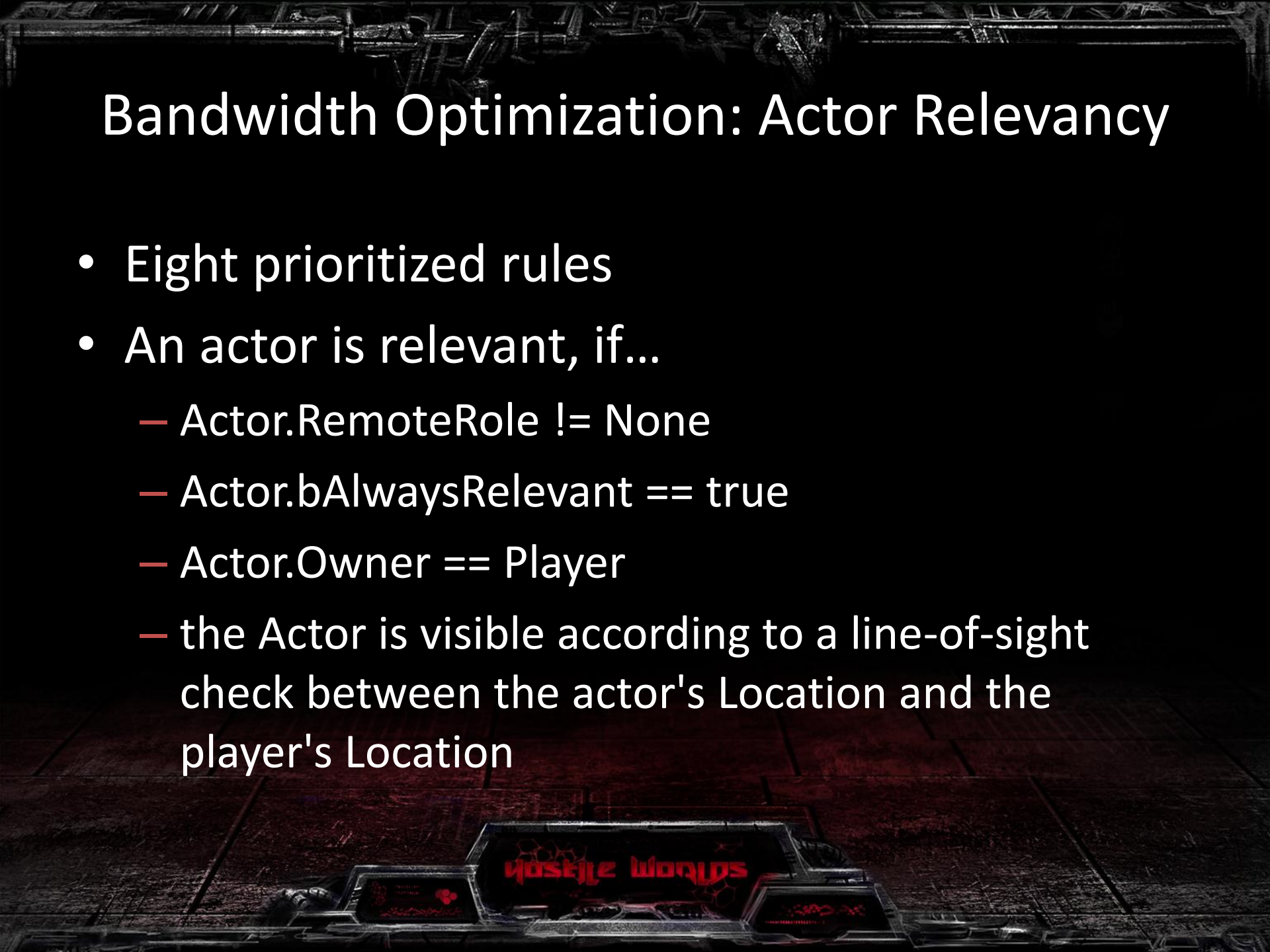
- Describes how much control the machine (server or client) has over an actor
- Controls the actors function call permissions

```
// Net variables.
enum ENetRole
{
    ROLE_None,              // No role at all.
    ROLE_SimulatedProxy,    // Locally simulated proxy of this actor.
    ROLE_AutonomousProxy,   // Locally autonomous proxy of this actor.
    ROLE_Authority,         // Authoritative control over the actor.
};

var ENetRole RemoteRole, Role;
```

Source: Actor.uc.

Bandwidth Optimization: Actor Relevancy

- Eight prioritized rules
- An actor is relevant, if…
  - Actor.RemoteRole != None
  - Actor.bAlwaysRelevant == true
  - Actor.Owner == Player
  - the Actor is visible according to a line-of-sight check between the actor's Location and the player's Location



Bandwidth Optimization: Actor Relevancy

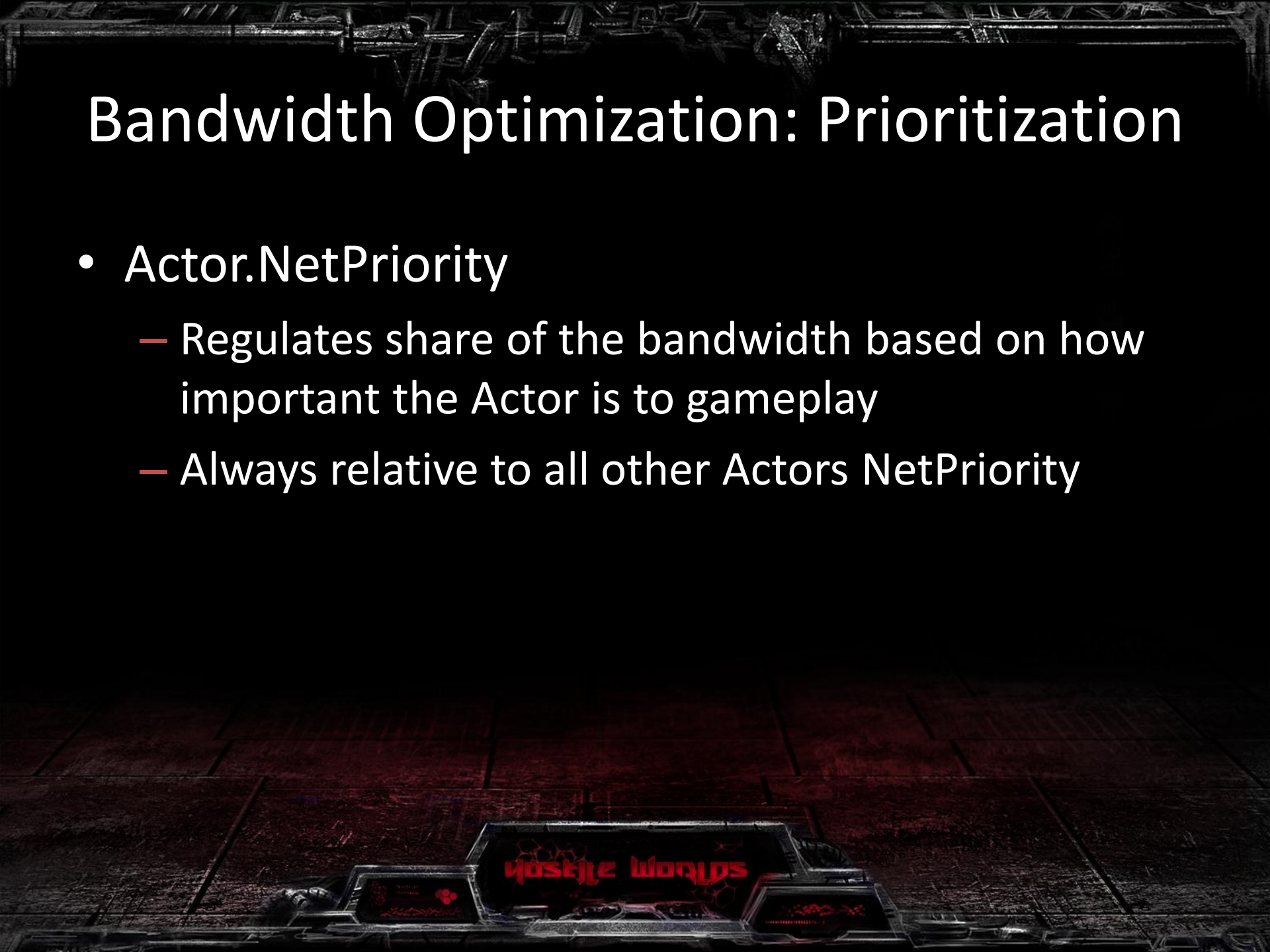
- Eight prioritized rules
- An actor is relevant, if…
  - Actor.RemoteRole != None
  - Actor.bAlwaysRelevant == true
  - Actor.Owner == Player
  - the Actor is visible according to a line-of-sight check between the actor's Location and the player's Location

Bandwidth Optimization: Actor Relevancy

- Eight prioritized rules
- An actor is relevant, if…
  - Actor.RemoteRole != None
  - Actor.bAlwaysRelevant == true
  - Actor.Owner == Player
  - the Actor is visible according to a line-of-sight check between the actor's Location and the player's Location

Bandwidth Optimization: Prioritization

- Actor.NetPriority
  - Regulates share of the bandwidth based on how important the Actor is to gameplay
  - Always relative to all other Actors NetPriority



Replication

- Actor Replication
  - Only Location, Rotation valid on spawn
- Variable Replication
  - Regulated by condition in Class Replication Statement
  - Server to Client only
  - Always reliable
  - Subject to bandwidth optimization
  - Keyword: `repnotify`

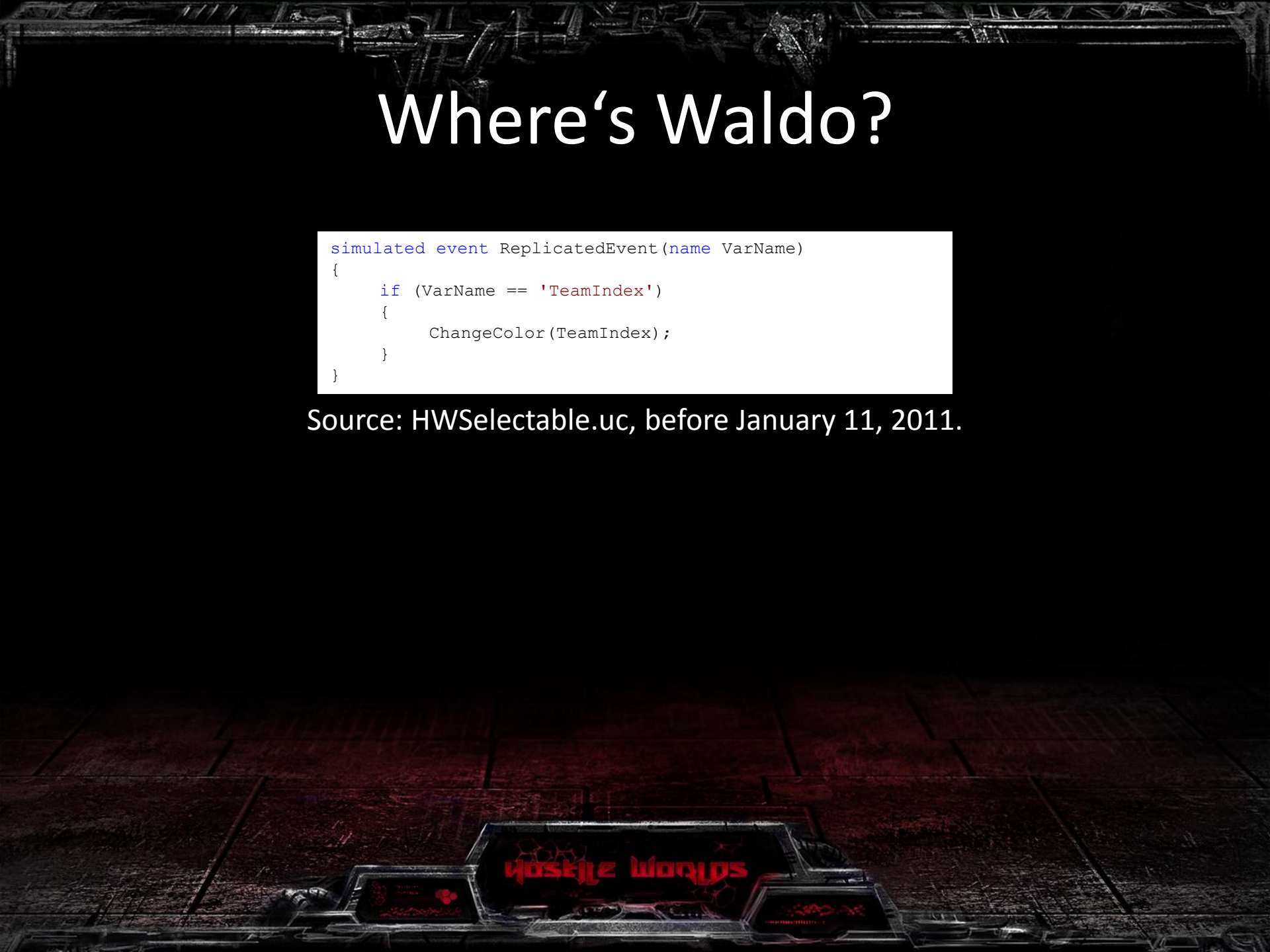
```
replication
{
     // replicate if server
     if (Role == ROLE_Authority && (bNetInitial || bNetDirty))
          Armor, Range, AttackDamage;
}
```

Source: HWPawn.uc.

Where‘s Waldo?

```
simulated event ReplicatedEvent(name VarName)
{
     if (VarName == 'TeamIndex')
     {
          ChangeColor(TeamIndex);
     }
}
```

Source: HWSelectable.uc, before January 11, 2011.

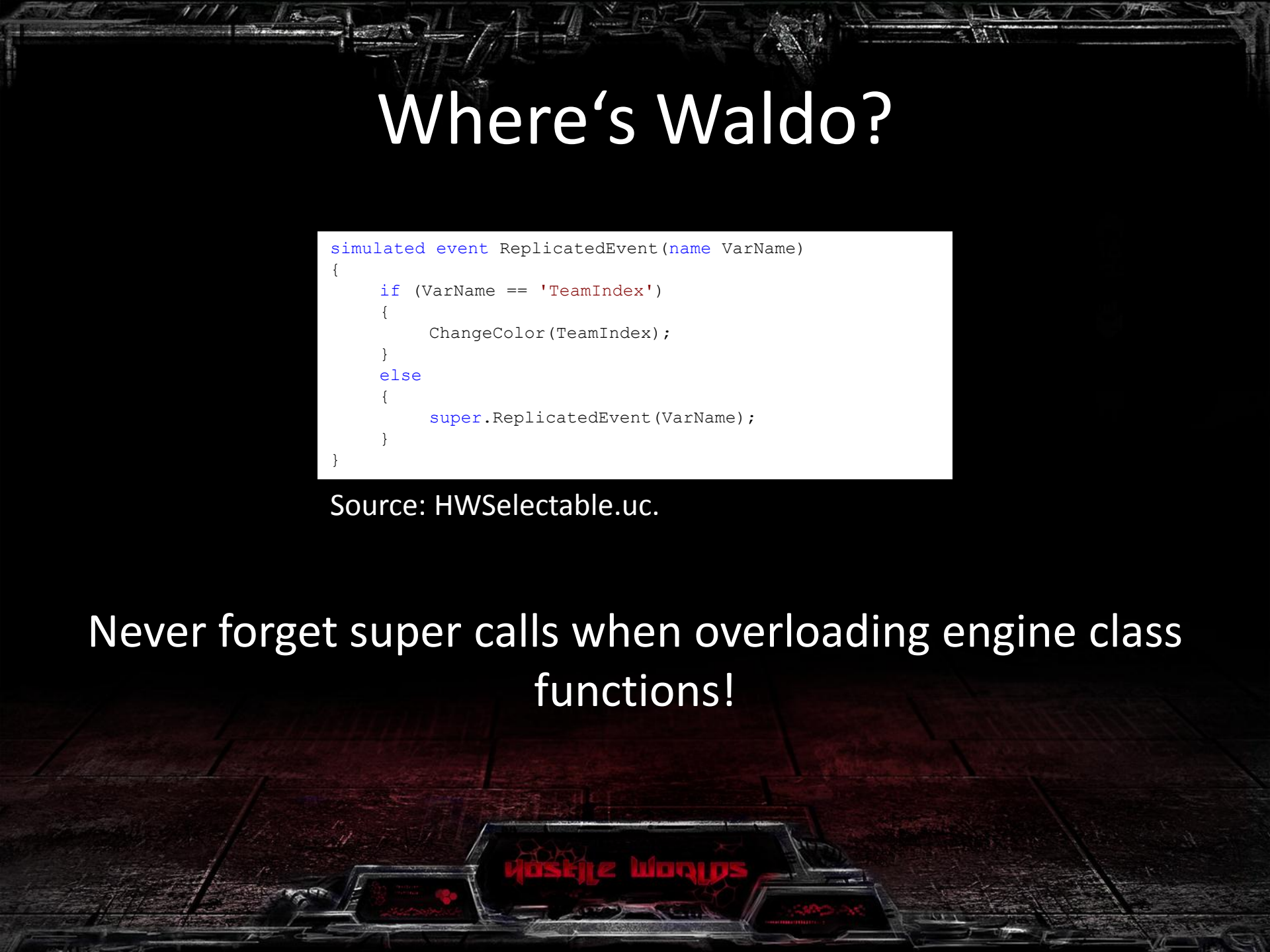


Where‘s Waldo?

```
simulated event ReplicatedEvent(name VarName)
{
     if (VarName == 'TeamIndex')
     {
          ChangeColor(TeamIndex);
     }
     else
     {
          super.ReplicatedEvent(VarName);
     }
}
```

Source: HWSelectable.uc.

Never forget super calls when overloading engine class functions!

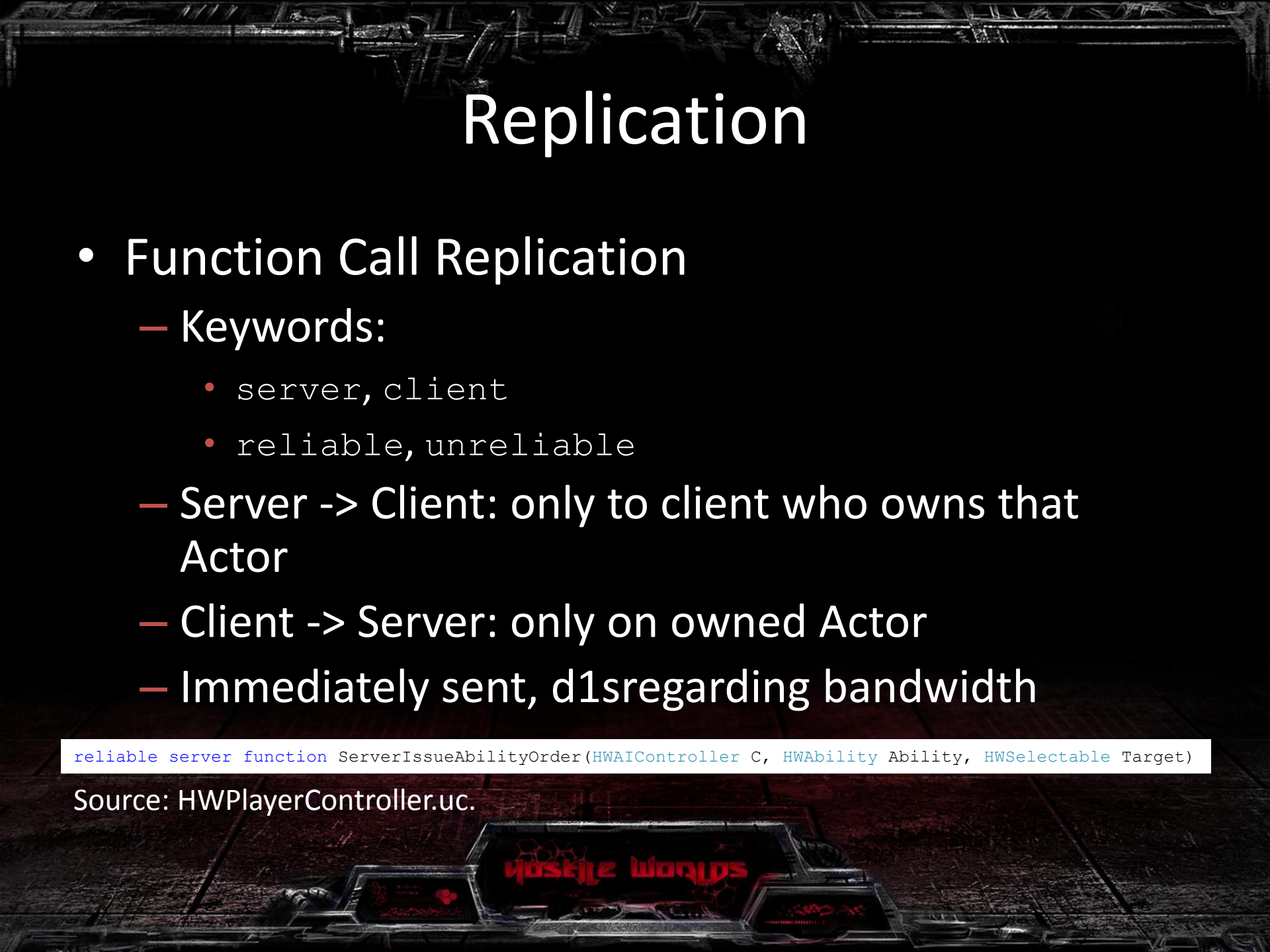


Replication

- Function Call Replication
  - Keywords:
    - `server`, `client`
    - `reliable`, `unreliable`
  - Server -> Client: only to client who owns that Actor
  - Client -> Server: only on owned Actor
  - Immediately sent, d1sregarding bandwidth

```
reliable server function ServerIssueAbilityOrder(HWAIController C, HWAbility Ability, HWSelectable Target)
```

Source: HWPlayerController.uc.



Fog of War

Fog of War in StarCraft II.

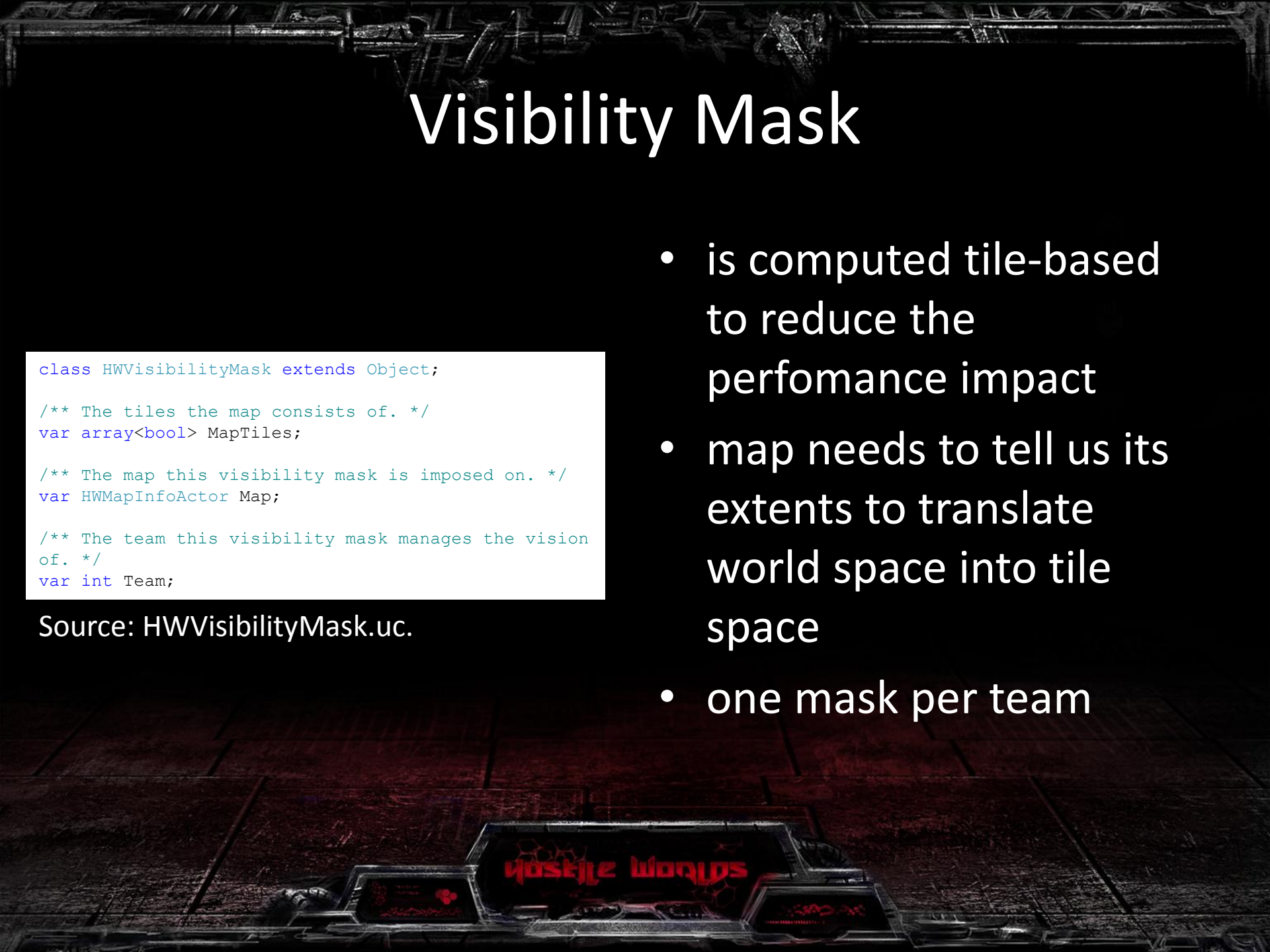
- hides any enemy units that aren‘t within the *sight radius* of a friendly unit
- needs to be computed efficiently

Visibility Mask

```
class HWVisibilityMask extends Object;

/** The tiles the map consists of. */
var array<bool> MapTiles;

/** The map this visibility mask is imposed on. */
var HWMapInfoActor Map;

/** The team this visibility mask manages the vision
of. */
var int Team;
```

Source: HWVisibilityMask.uc.

- is computed tile-based to reduce the perfomance impact
- map needs to tell us its extents to translate world space into tile space
- one mask per team



Visibility Mask

```
class HWVisibilityMask extends Object;

/** The tiles the map consists of. */
var array<bool> MapTiles;

/** The map this visibility mask is imposed on. */
var HWMapInfoActor Map;

/** The team this visibility mask manages the vision
of. */
var int Team;
```

Source: HWVisibilityMask.uc.

- is computed tile-based to reduce the perfomance impact
- map needs to tell us its extents to translate world space into tile space
- one mask per team

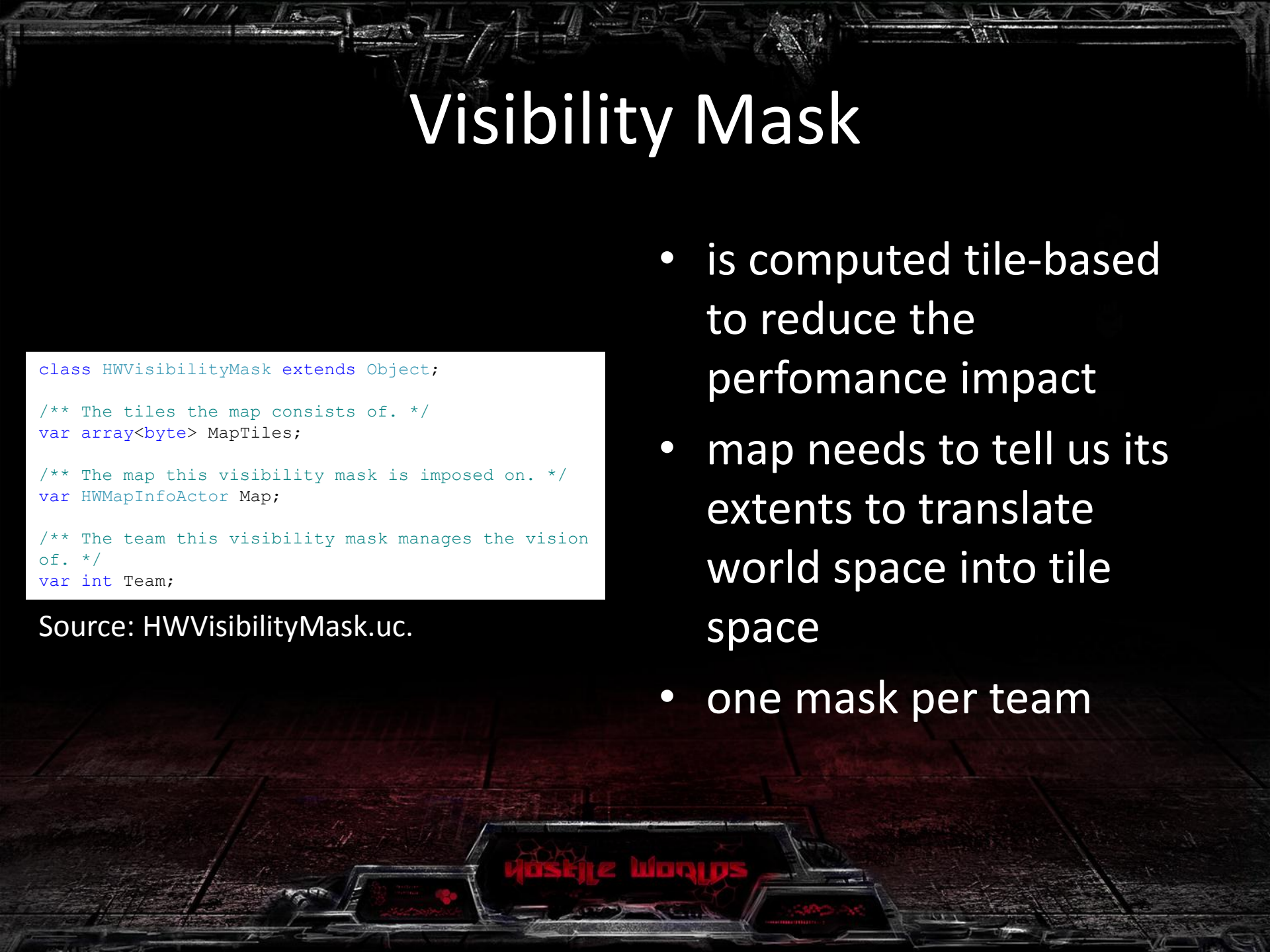
```
Src\HostileWorlds\Classes\HWVisibilityMask.uc(14) : Error, Bool arrays are not allowed
Failure - 1 error(s), 0 warning(s)
```

Visibility Mask

```
class HWVisibilityMask extends Object;

/** The tiles the map consists of. */
var array<byte> MapTiles;

/** The map this visibility mask is imposed on. */
var HWMapInfoActor Map;

/** The team this visibility mask manages the vision
of. */
var int Team;
```

Source: HWVisibilityMask.uc.

- is computed tile-based to reduce the perfomance impact
- map needs to tell us its extents to translate world space into tile space
- one mask per team



Updating the Visibility Mask

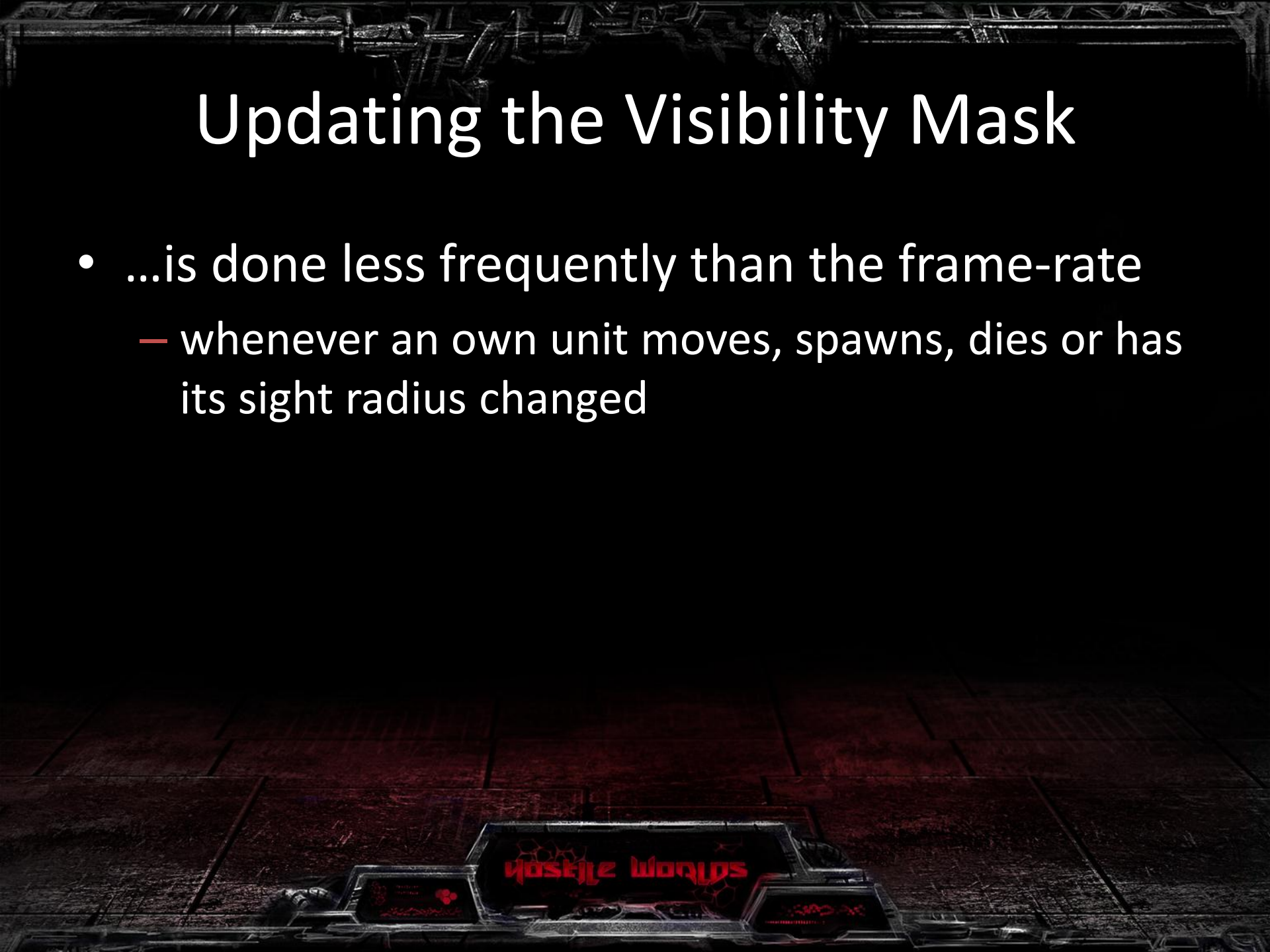
```
/** Updates this visibility mask, re-computing the vision for
the team this mask belongs to. */
simulated function Update()
{
    local HWSelectable s;
    local IntPoint Tile;
    local array<IntPoint> Tiles;

    // reset visibility
    foreach Map.DynamicActors(class'HWSelectable', s)
    {
        if (s.TeamIndex == Team)
        {
            HideMapTiles(s);
        }
    }

    // compute new visibility
    foreach Map.DynamicActors(class'HWSelectable', s)
    {
        if (s.TeamIndex == Team)
        {
            Tile = Map.GetMapTileFromLocation(s.Location);
            Tiles = Map.GetListOfCircleTiles
                (Tile, s.SightRadiusTiles);
            RevealMapTiles(Tiles, s);
        }
    }
}
```

Source: HWVisibilityMask.uc

1. reset mask
   - no memset in Unreal: units remember the tiles they can see
2. compute new mask
   1. iterate team‘s units
   2. translate their positions into tile space
   3. compute sight circle
   4. reveal tiles

Updating the Visibility Mask

- …is done less frequently than the frame-rate
  - whenever an own unit moves, spawns, dies or has its sight radius changed



Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Translating World Space into Tile Space

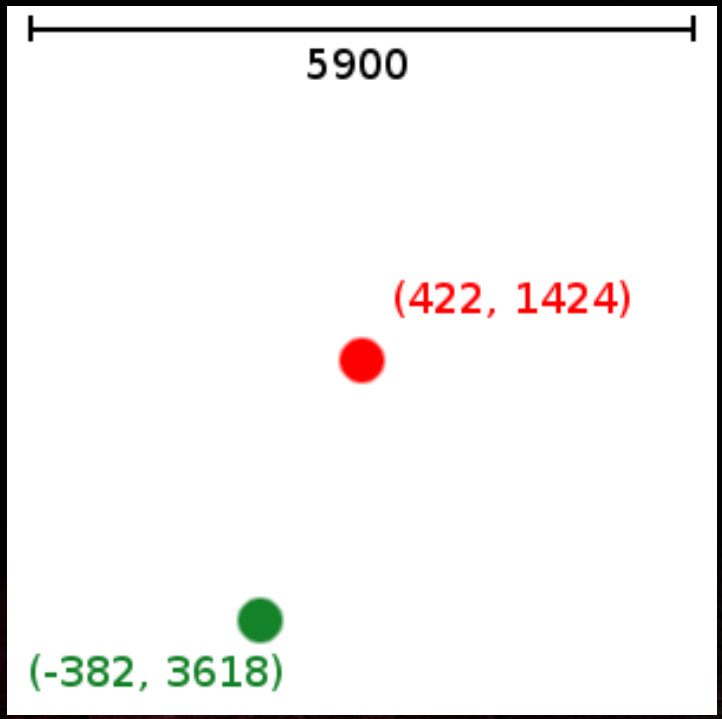
$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Maybe we should take a closer look…

Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

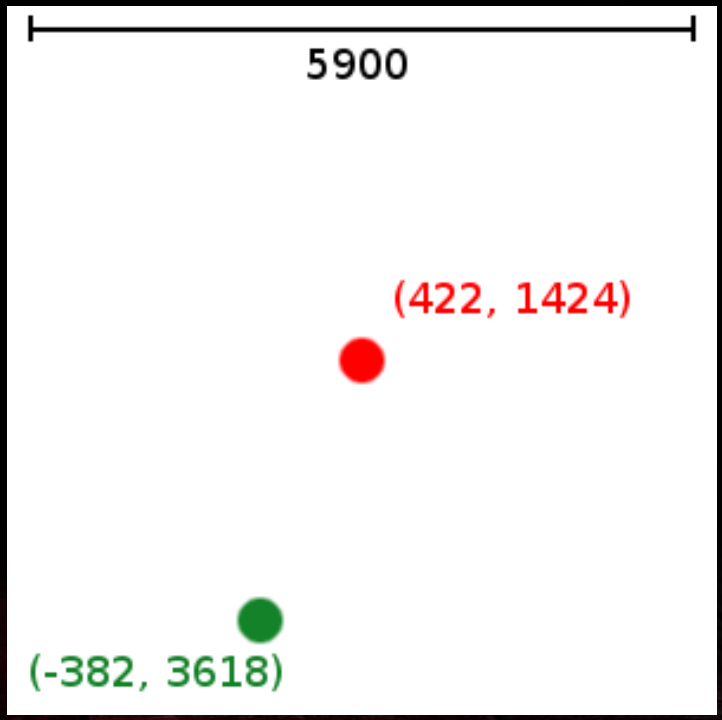
- in
  - position of an object in world space
  - position of the map center in world space
  - width and height of the map, in UU
  - width and height of the map, in tiles
- out
  - position of the object in tile space



Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

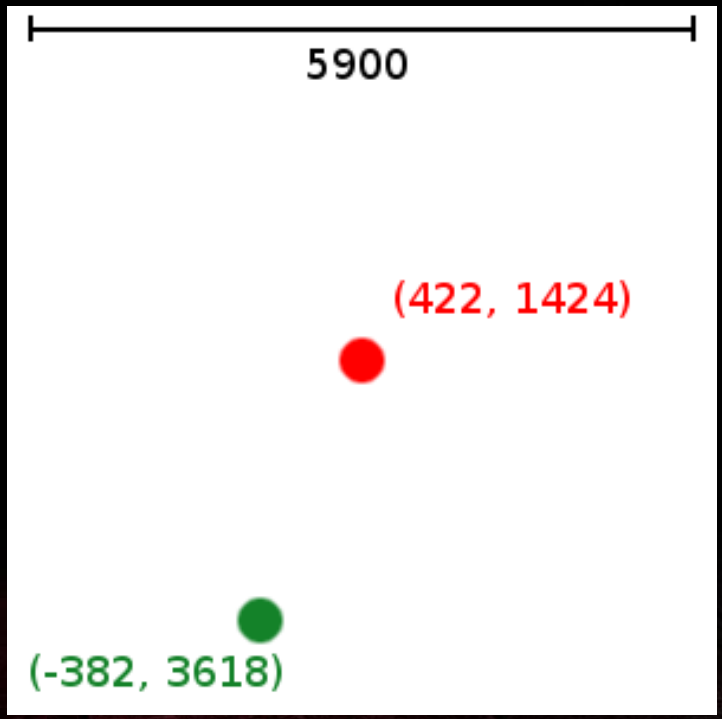
- in
  - position of an object in world space
  - position of the map center in world space
  - width and height of the map, in UU
  - width and height of the map, in tiles
- out
  - position of the object in tile space



Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

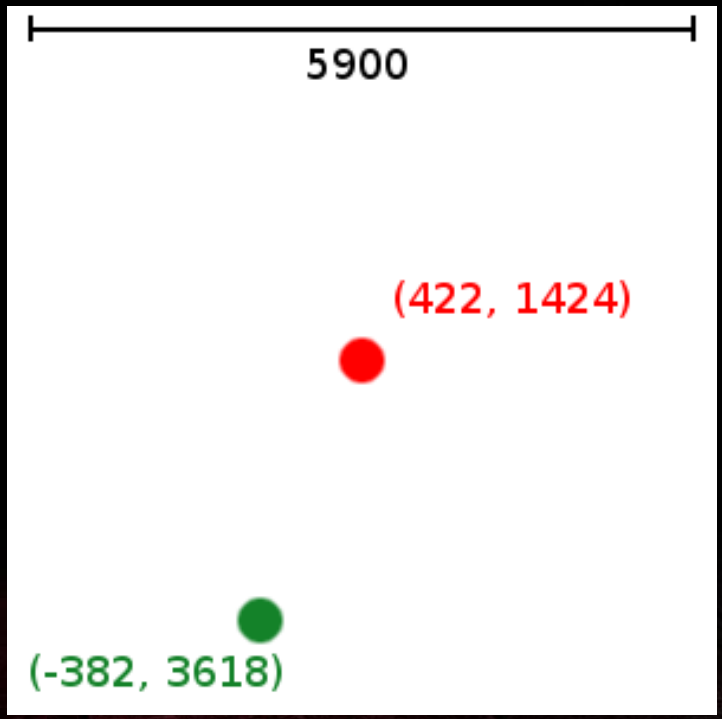
- in
  - position of an object in world space
  - position of the map center in world space
  - width and height of the map, in UU
  - width and height of the map, in tiles
- out
  - position of the object in tile space



Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

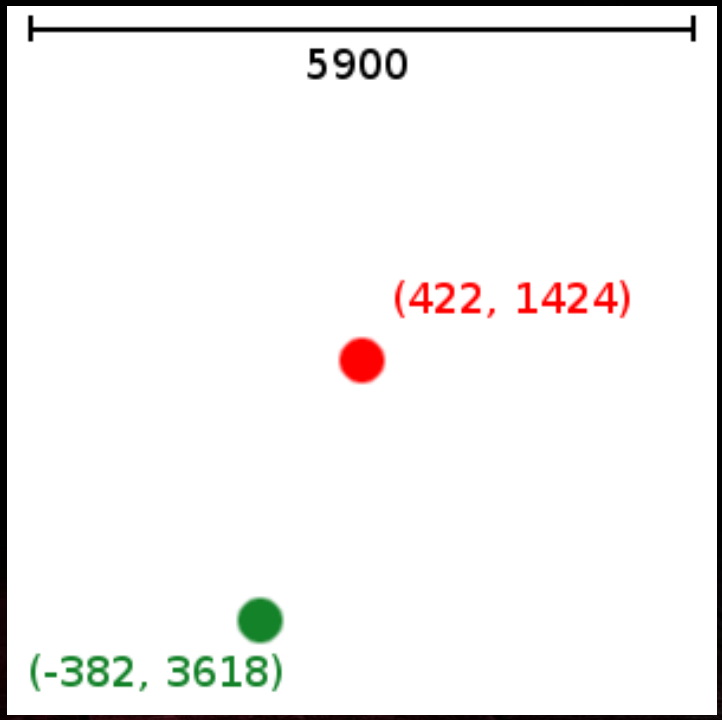
- in
  - position of an object in world space
  - position of the map center in world space
  - width and height of the map, in UU
  - width and height of the map, in tiles
- out
  - position of the object in tile space



Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

- in
  - position of an object in world space
  - position of the map center in world space
  - width and height of the map, in UU
  - width and height of the map, in tiles
- out
  - position of the object in tile space

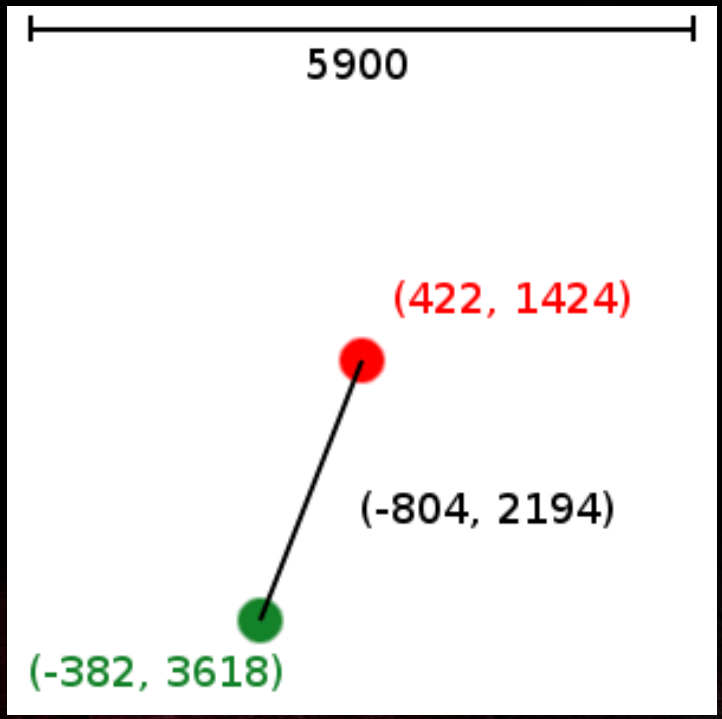


Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Step 1:

Compute the object‘s offset from the map center.

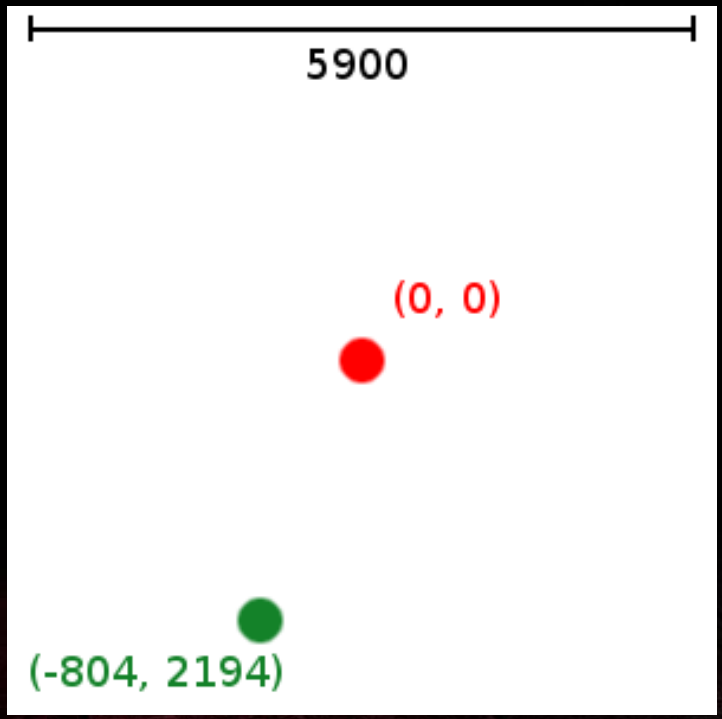


Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Step 2:

Transform the object‘s position into offset space.

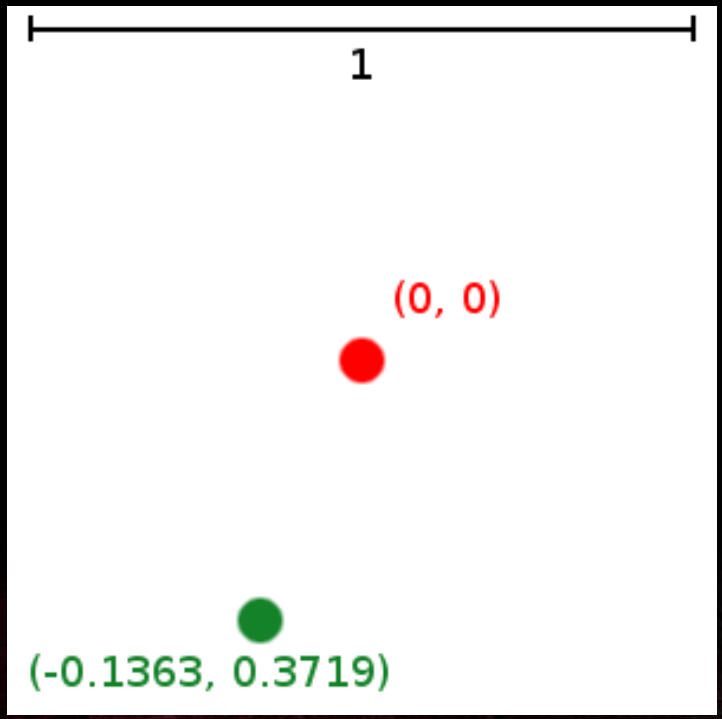


Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Step 3:

Normalize the object‘s offset by dividing by the map dimension.

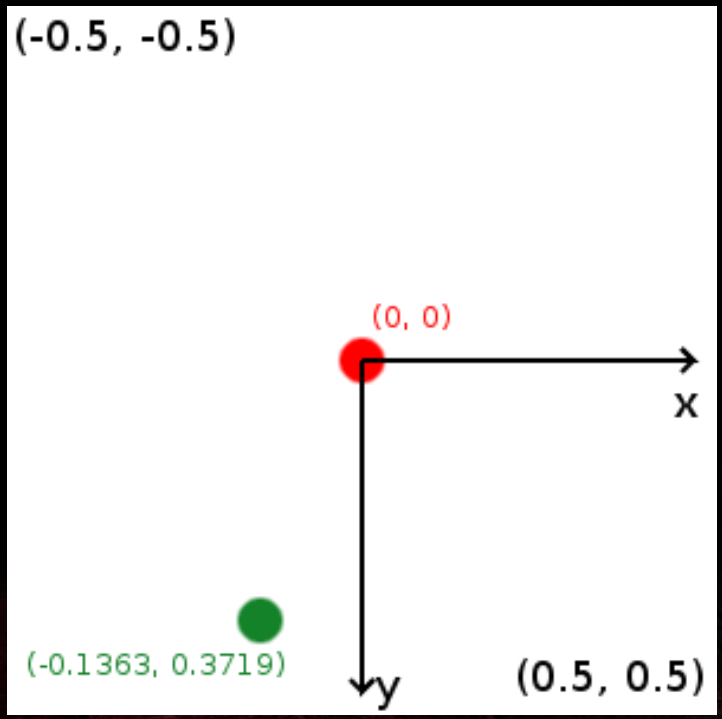


Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Step 3:

Normalize the object‘s offset by dividing by the map dimension.

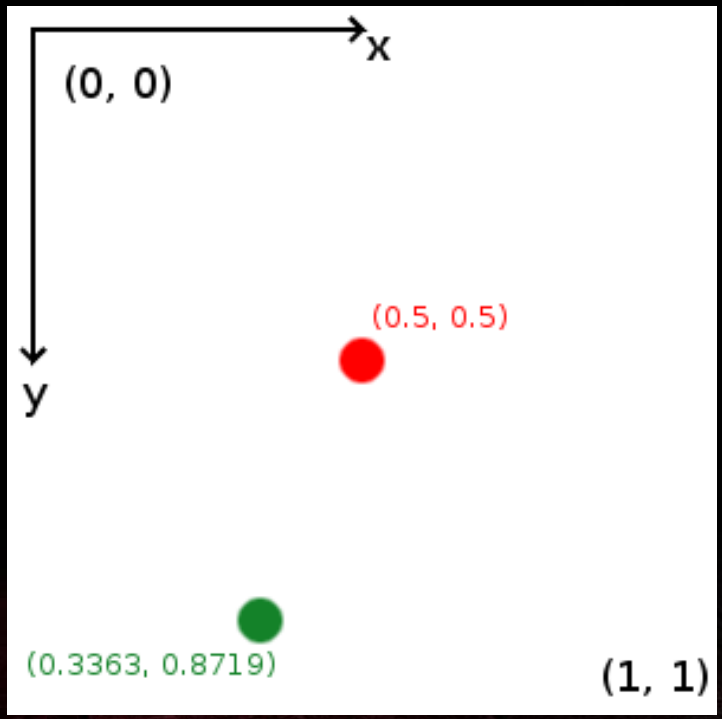


Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Step 4:

Translate the coordinate system by moving the origin to the upper left corner of the map.

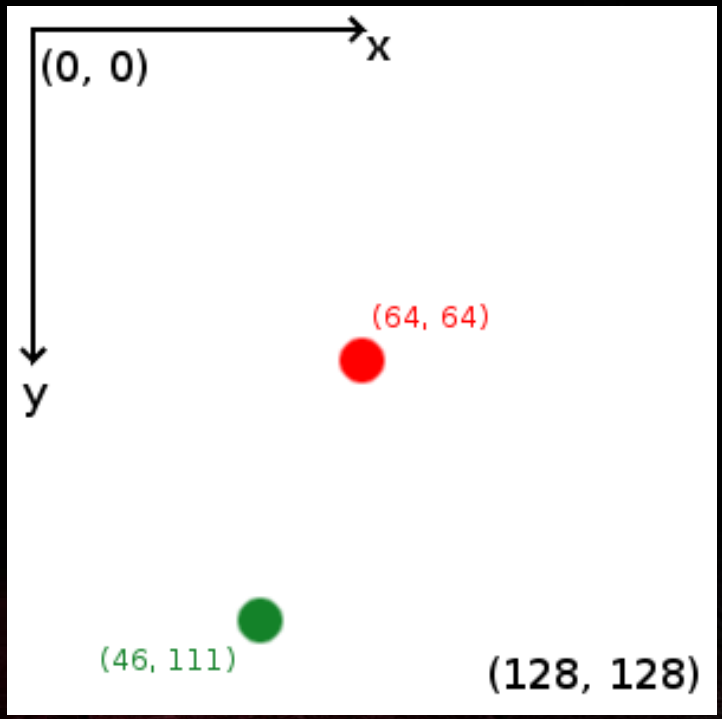


Translating World Space into Tile Space

$$\overrightarrow{Pos_{Tile}} = \left\lfloor \left(\frac{\overrightarrow{Pos_{World}} - \overrightarrow{MapCenter}}{MapDim} + 0.5 \right) \cdot Tiles \right\rfloor$$

Step 5:

Compute the position in tile space by multiplying with the number of tiles, rounding down.

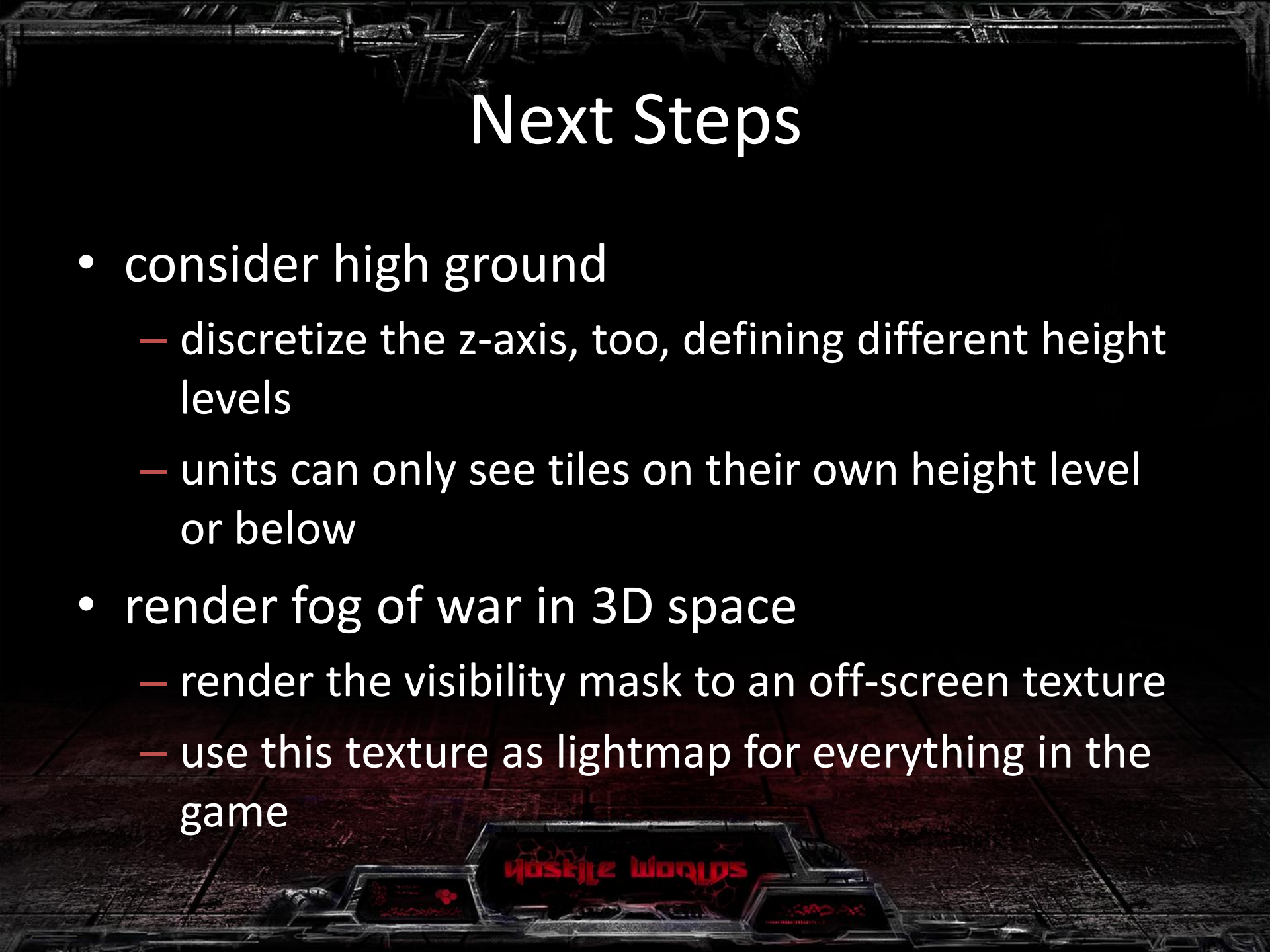


Applying Fog of War Logic

- iterate all units not belonging to the own team
  - translate their position into tile space
  - check whether the tile is visible in the visibility mask
- hide and deselect all enemy units that get hidden by fog of war
- cancel all orders of the own team targeting these units
- do server-side checks for all orders or abilities

Next Steps

- consider high ground
  - discretize the z-axis, too, defining different height levels
  - units can only see tiles on their own height level or below
- render fog of war in 3D space
  - render the visibility mask to an off-screen texture
  - use this texture as lightmap for everything in the game



Minimap

The minimap of StarCraft II.

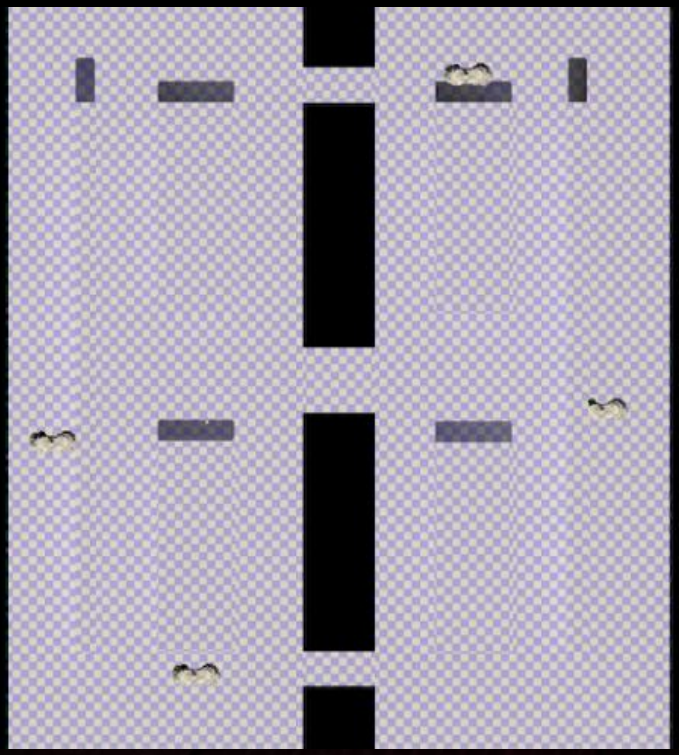
- gives the player an overview of the entire map at a glance
- allows issuing orders targeting locations outside the current view frustrum

Minimap

The minimap of StarCraft II.

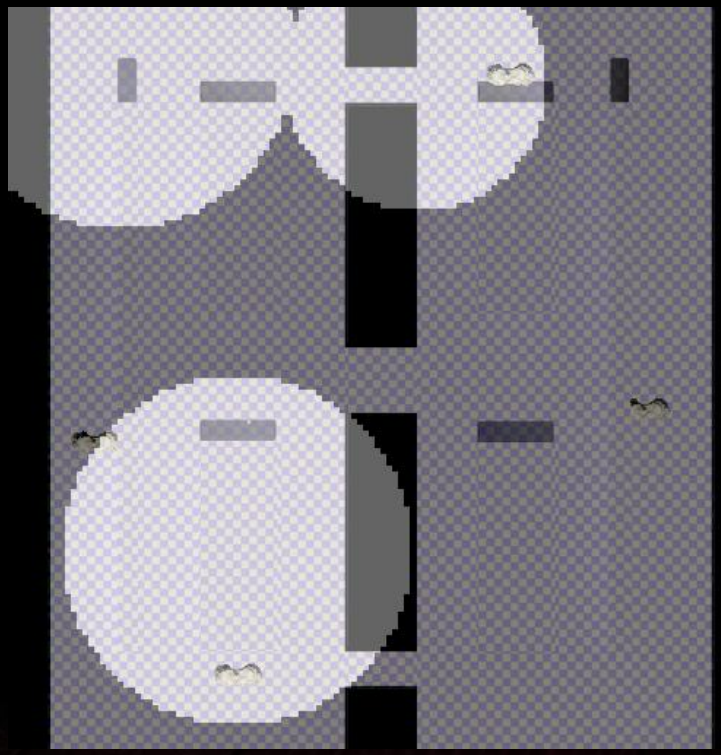
- consists of three layers:
  1. landscape layer
  2. fog of war layer
  3. unit layer
- all are rendered to different textures that are blended together

Minimap: Landscape Layer



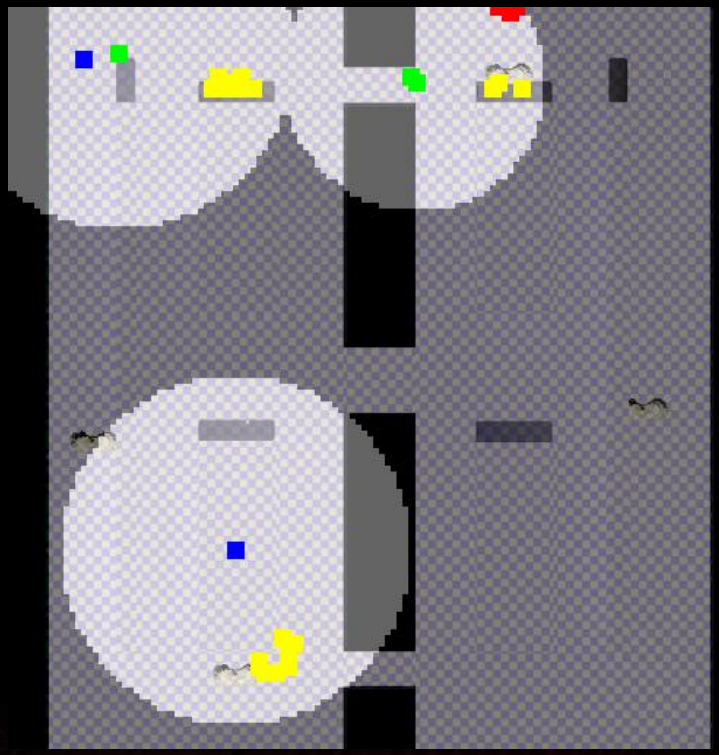
- orthogonal rendering of the terrain without any camera culling
- updated when the terrain itself changes only

Minimap: Fog of War Layer



- visibility mask is rendered to an off-screen texture
- updated whenever the visibility is updated

Minimap: Unit Layer



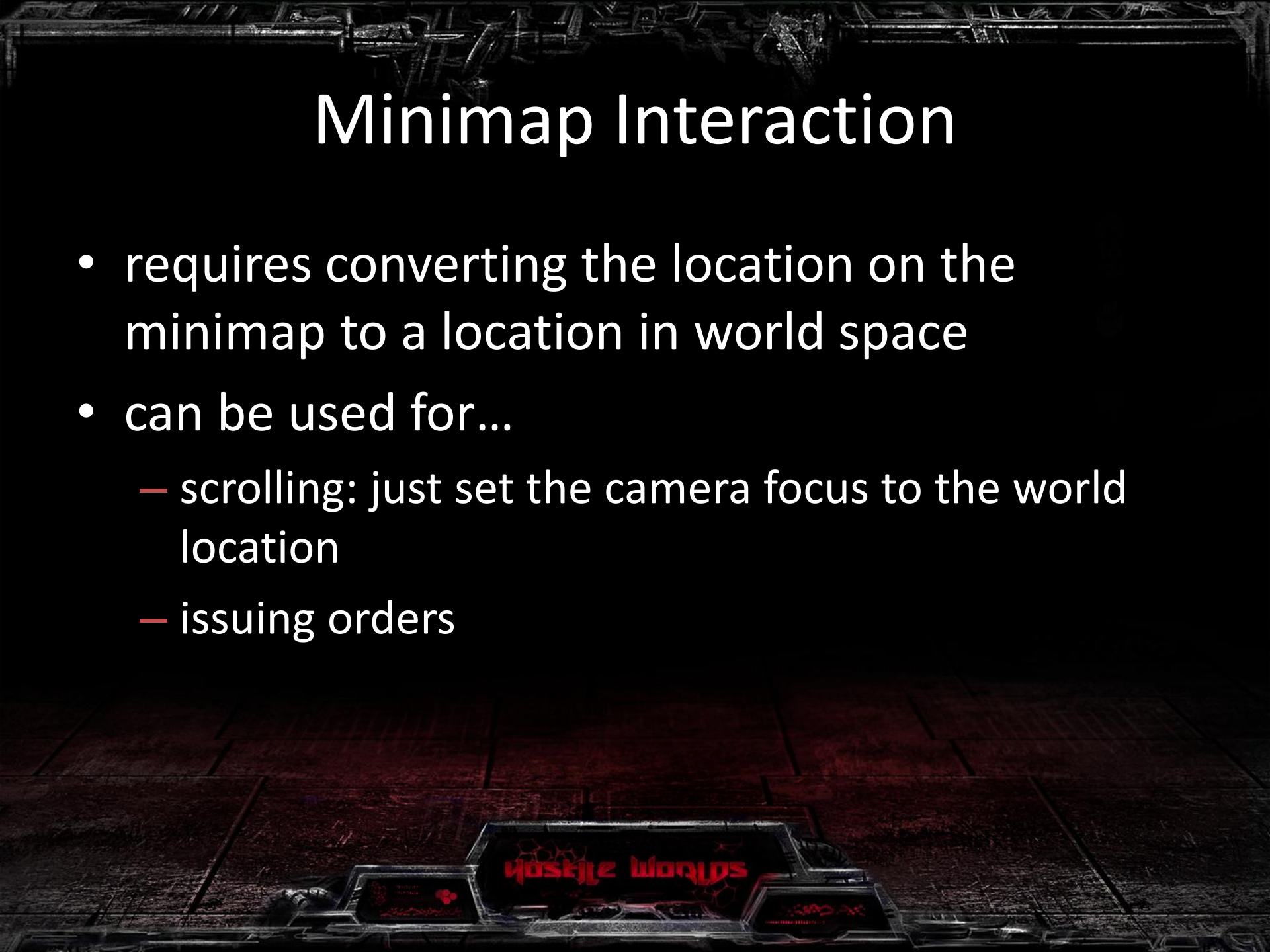
- shows the positions and owners of all visible units and game objects
- updated whenever a unit moves from one map tile to another

Minimap: View Frustum

- shows the player which part of the map he or she is looking at
- can be drawn as follows:
  - cast a ray from the eye of the camera to each of the frustum corners in the far plane
  - translate the point of their intersection with the terrain from world space to minimap space
  - draw lines between the four corners on the minimap

Minimap Interaction

- requires converting the location on the minimap to a location in world space
- can be used for…
  - scrolling: just set the camera focus to the world location
  - issuing orders



Bibliography

1. Carl Granberg. Programming an RTS Game with Direct3D. Charles River Media, pages 265-307, 2007.
2. http://udn.epicgames.com/Three/UDKProgrammingHome.html
3. Unreal Development Kit (August 2010) Source Code

Thank you for your attention!

www.hostile-worlds.com