Threading Game Engines

- QUAKE 4 & Enemy Territory QUAKE Wars

Anu Kalra - Intel Corporation

Jan Paul van Waveren - id Software

Feb 21, 2006

Agenda

- Concurrency In Games Today
- Analysis of QUAKE 4
- Renderer Threading QUAKE 4 and ETQW
- AI & Mega texture Threading in ETQW
- Common Performance Issues & Workarounds
- Building Scalability into threading design

Concurrency In Games

- There has been a dramatic increase in compute power in consumer space in the last few years with multi-core
  - Game industry has started the move to adopt concurrent programming
- Most multithreaded games today still follow the first generation of parallelism i.e. threading based on functional decomposition.
- Game is broken up into various subsystems each of which run on their own thread typically rendering, and AI sometimes physics too


QUAKE 4

QUAKE 4 Analysis

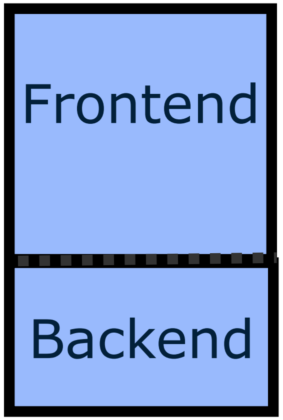
- As per the V-tune Analysis QUAKE 4 was
  - CPU bound
  - Predominantly Single threaded
  - Roughly equal amount is being spent in the driver and the engine 41% & 49% respectively
  - Each of the major hotspots consume 2-4% of CPU time.
- Peek into the source revealed
  - QUAKE 4 had a good separation between the renderer Front and Back end.
  - Most of the time spent in the OpenGL driver came from the Renderer Backend.

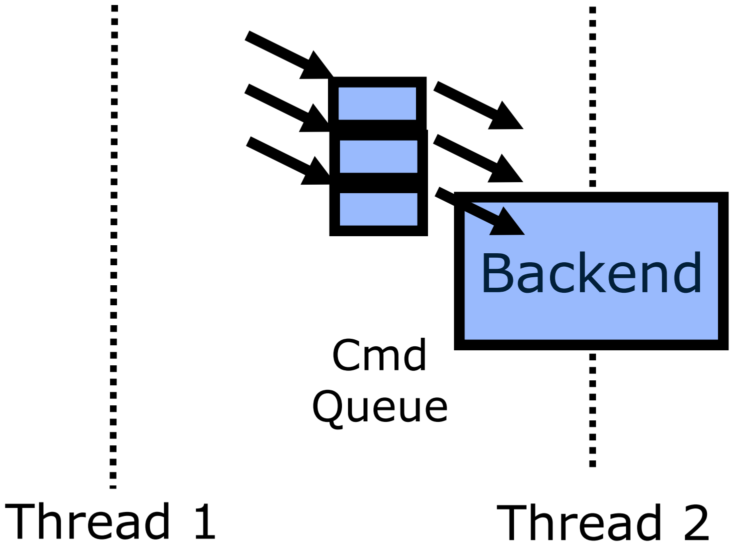
Constraints

– Threading an existing engine
– Time frame 4-6 months
– Target platform – P4 dual core (3.2 Ghz)
– Single core performance difference had to be less than 5%

Threading

- To get most performance in a constrained time decided to functionally decompose the 2 largest blocks.
- Split the render into front-end and back-end
- The backend was made to run on its own thread
- The front-end and back-end communicated through command queues and synchronization events

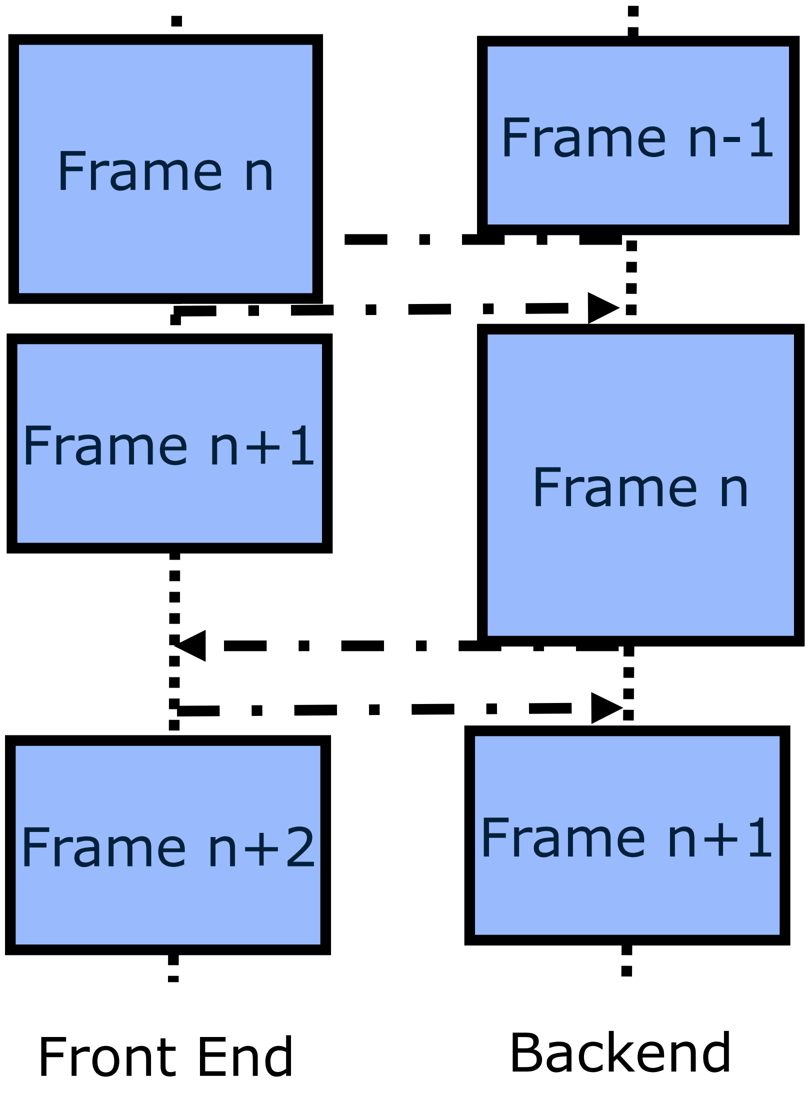




Threading

- The frame was prepared by the front end handed over to the back end while the front end prepared the next frame.
  - Data specific to a frame was double buffered
  - Data had to be allocated and freed safely.
  - Front end managed allocation & deallocation of shared data. Data to be freed was kept till the backend was done and cleared at the front end just before reuse.
  - Subsystems that were not thread safe had to be made thread safe models classes, animation, shadows, texture subsystems, deforms, loaders, writers, vertex caches, …

Synchronization

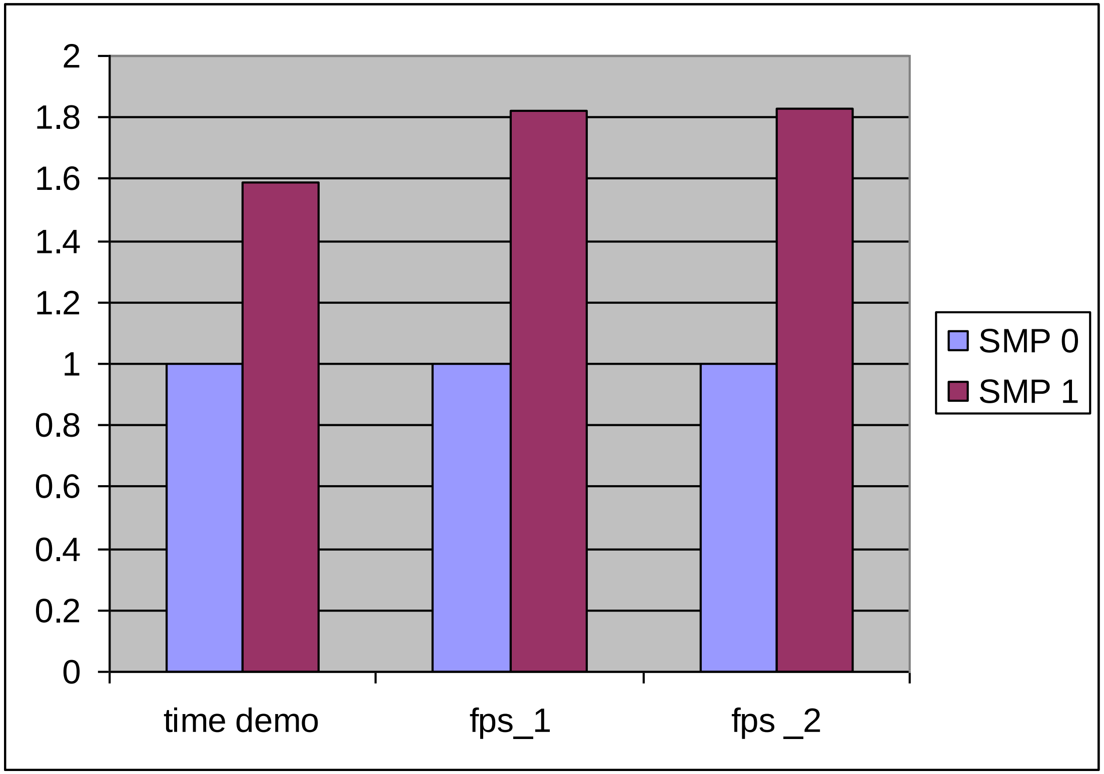


Issues with Threading

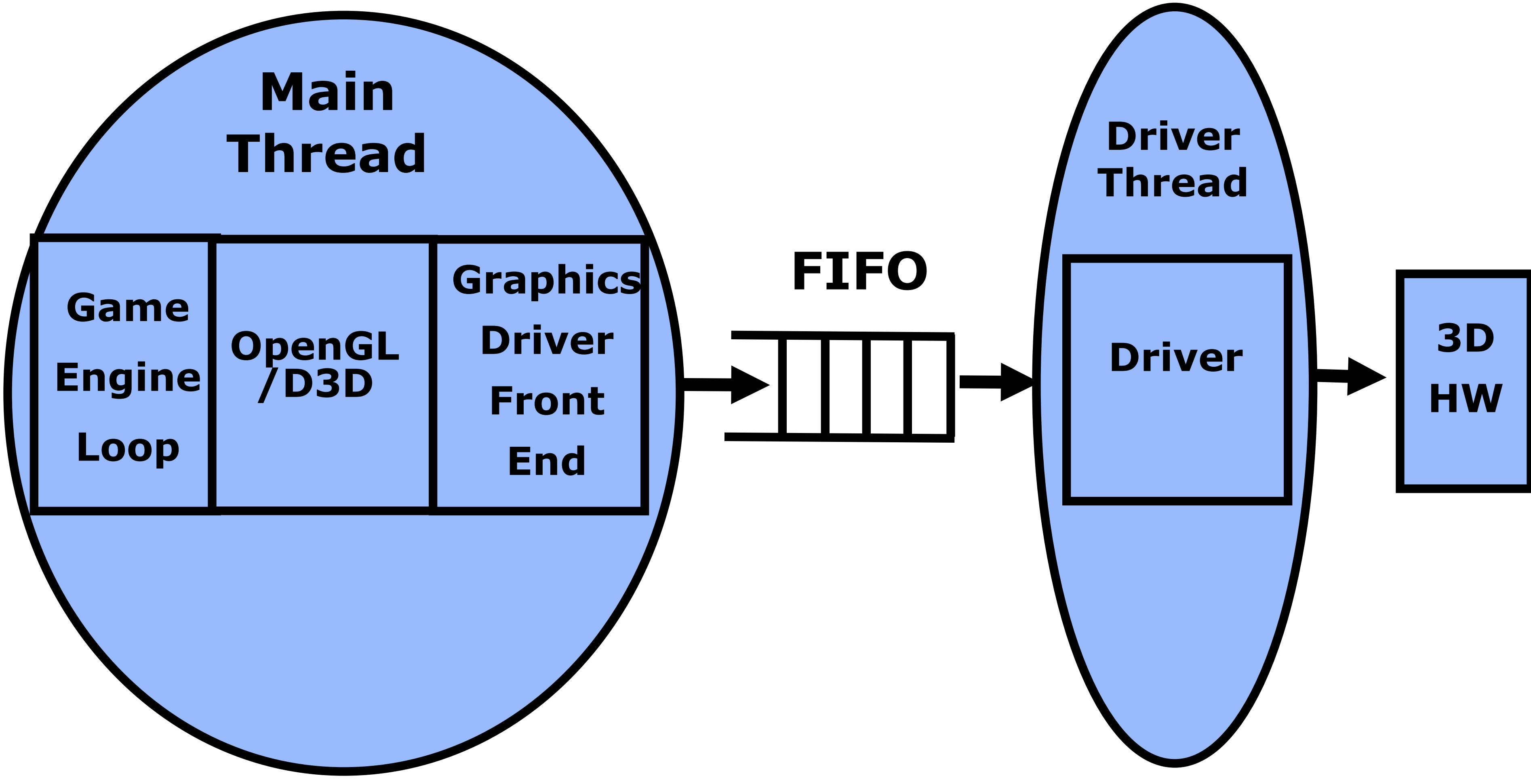
- Debugging The threaded code was the hardest
- Issues could be broadly categorized into 3 major types
  - Data Race Conditions
  - Object lifetime issues
  - OpenGL context issues
- Moved all the time critical OpenGL calls to the backend used a synch mechanism for others
- Added a realtime toggle capaility to turn threading on and off along with a lock step mode to the threaded code where the front end and back end would run on separate threads but run lock step
- Used Synchronization points to slowly & painfully eliminate Data Races
- Added lots of initialization and destruction code to deal with lifetime issues
- Needed to batch certain commands to improve performance

Performance Improvements

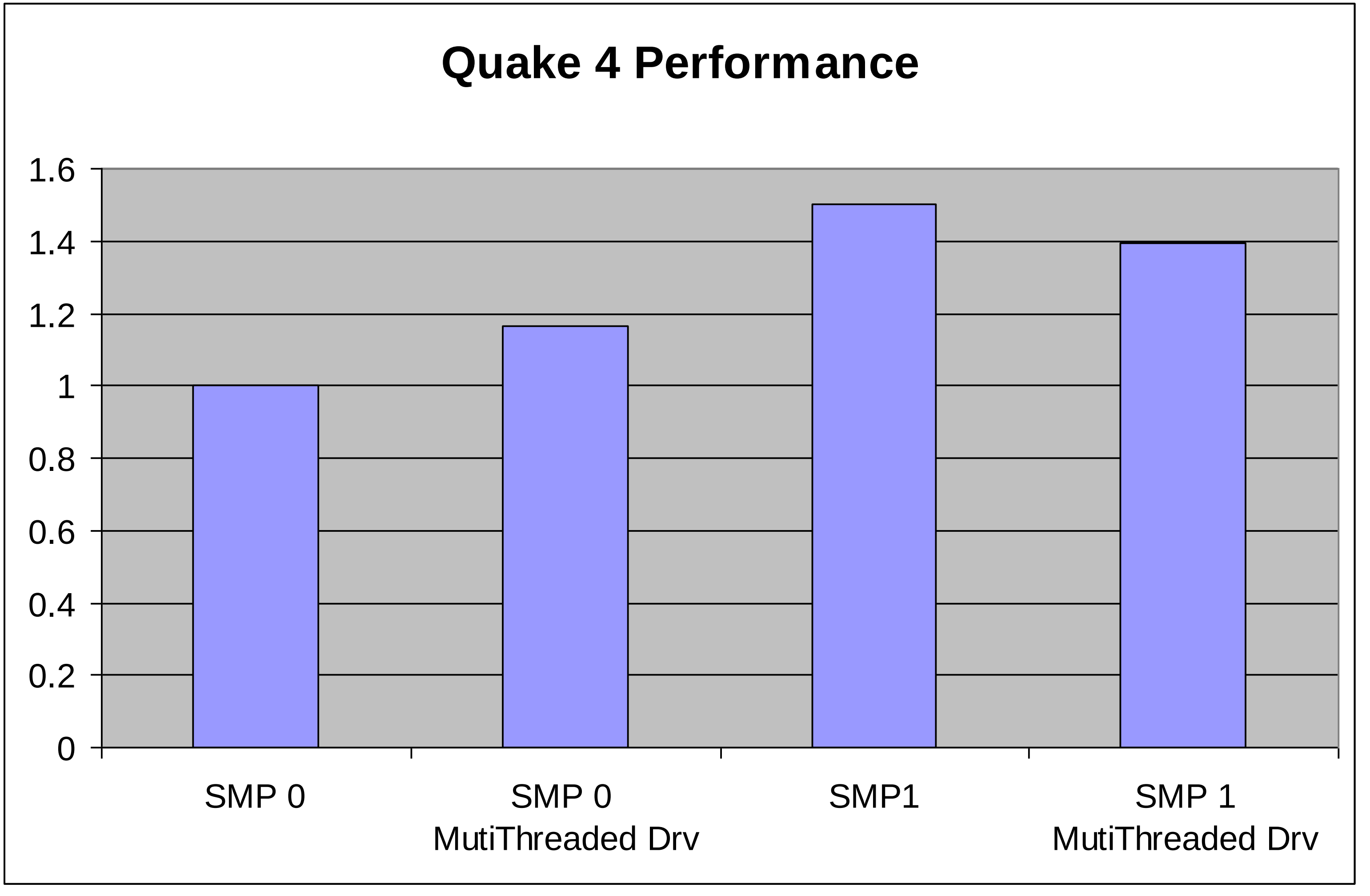
- Beta timeframe



Multi-Threaded Drivers



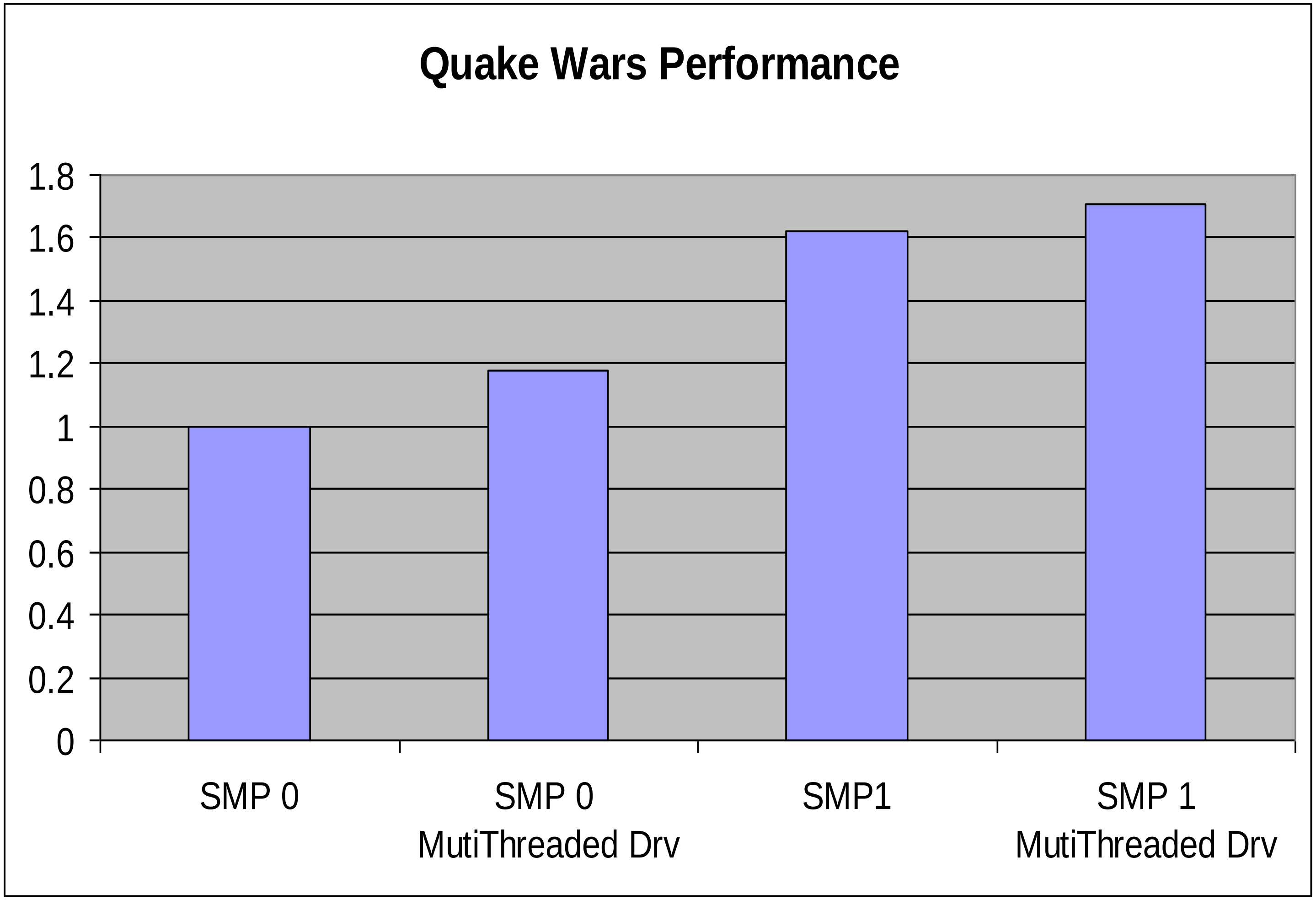
Current Performance



Renderer Threading with ETQW

- The whole renderer runs in a separate thread
- More work being done on the renderer thread
  - Culling and shadow volume construction
- Reduces amount of memory being buffered and shared between threads
  - Triangle meshes are not double buffered
- Better splitting of work on 2 cores
- Works better with multi-threaded drivers

ETQW



Quake Wars Performance
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
SMP 0
SMP 0 MutiThreaded Drv
SMP1
SMP 1 MutiThreaded Drv

Quake III Arena

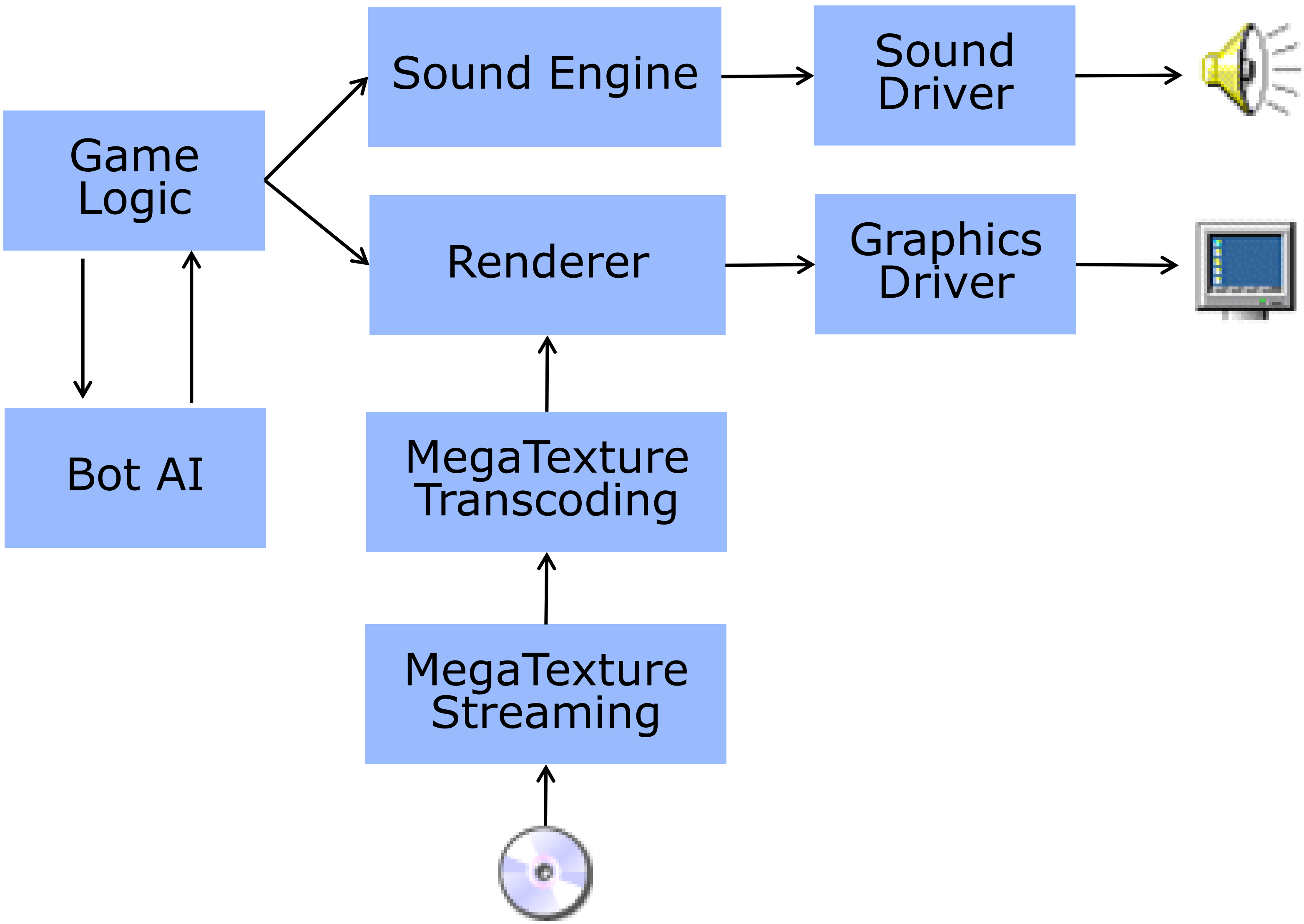
- Renderer back-end runs in a separate thread
- Very similar to QUAKE 4

DOOM III

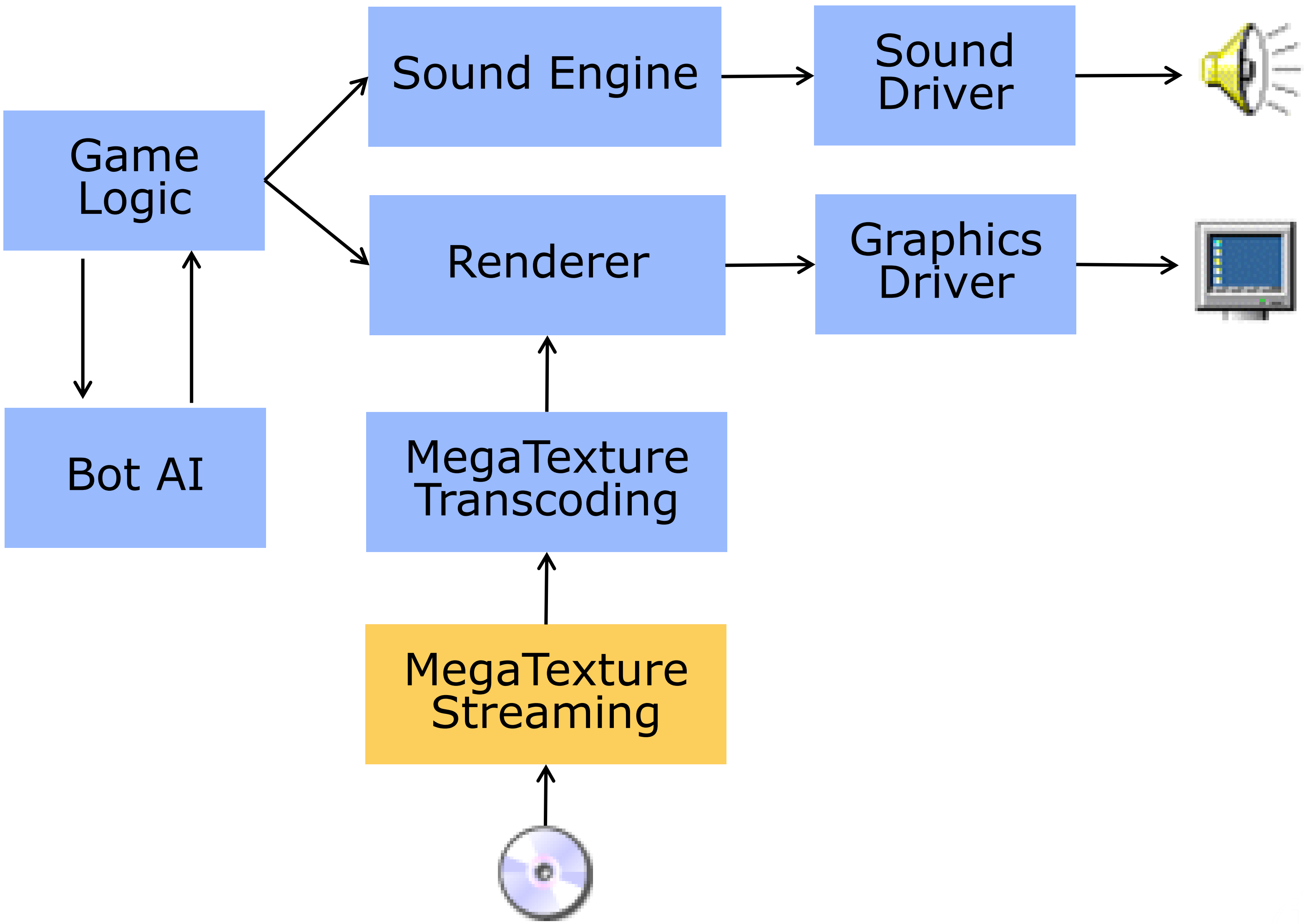
- Initially had the same threading as Quake III Arena
- Very much memory bound
- We actually removed the threading
- Instead SIMD optimized rendering pipeline
- The pipeline is optimized for cache usage

http://softwarecommunity.intel.com/articles/eng/2773.htm

ETQW Threading overview



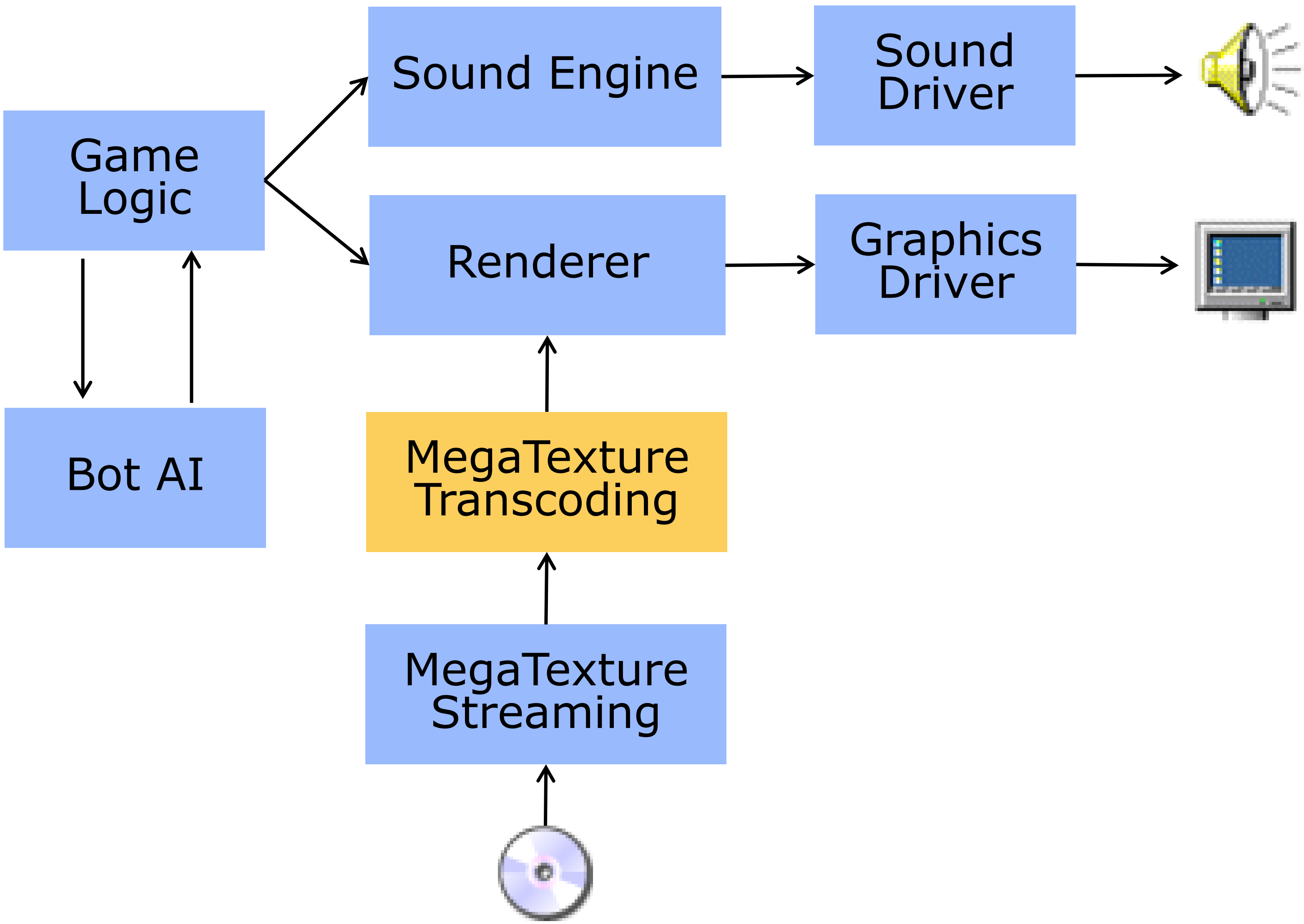
Mega Texture Streaming



Mega Texture Streaming

- The Mega Texture streaming thread dynamically sorts tile read requests.
- This thread is not doing any significant amount of work and mostly waits in place while data is being read from disk.
- The streaming is optimized using a texture database with an optimized layout to minimize seek times.
- The streaming thread reads 128 kB non-cached sector aligned blocks of data for optimal streaming from a DVD without polluting file system caches.

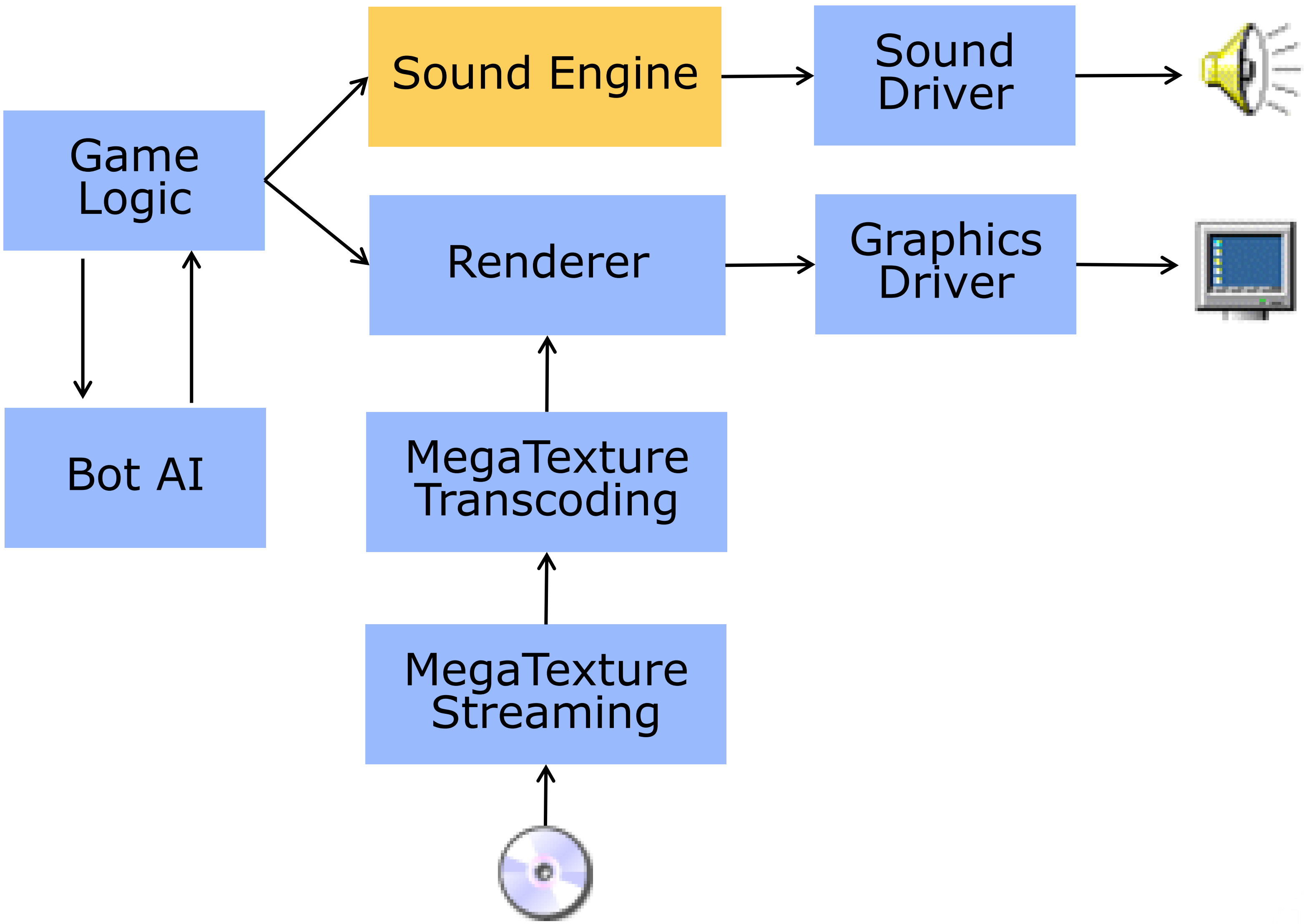
Mega Texture Transcoding



Mega Texture Transcoding

- Real-Time conversion from JPEG-like format to DXT.
- The transcoding uses highly optimized SIMD code and as such this thread does not consume a whole lot of CPU time.
- On systems based on the Core 2 microarchitecture the mega texture transcoding thread typically consumes less than 15% CPU time.
- Real-Time Texture Streaming & Decompression
  http://softwarecommunity.intel.com/articles/eng/1221.htm
- Real-Time DXT Compression
  http://www.intel.com/cd/ids/developer/asmo-na/eng/324337.htm

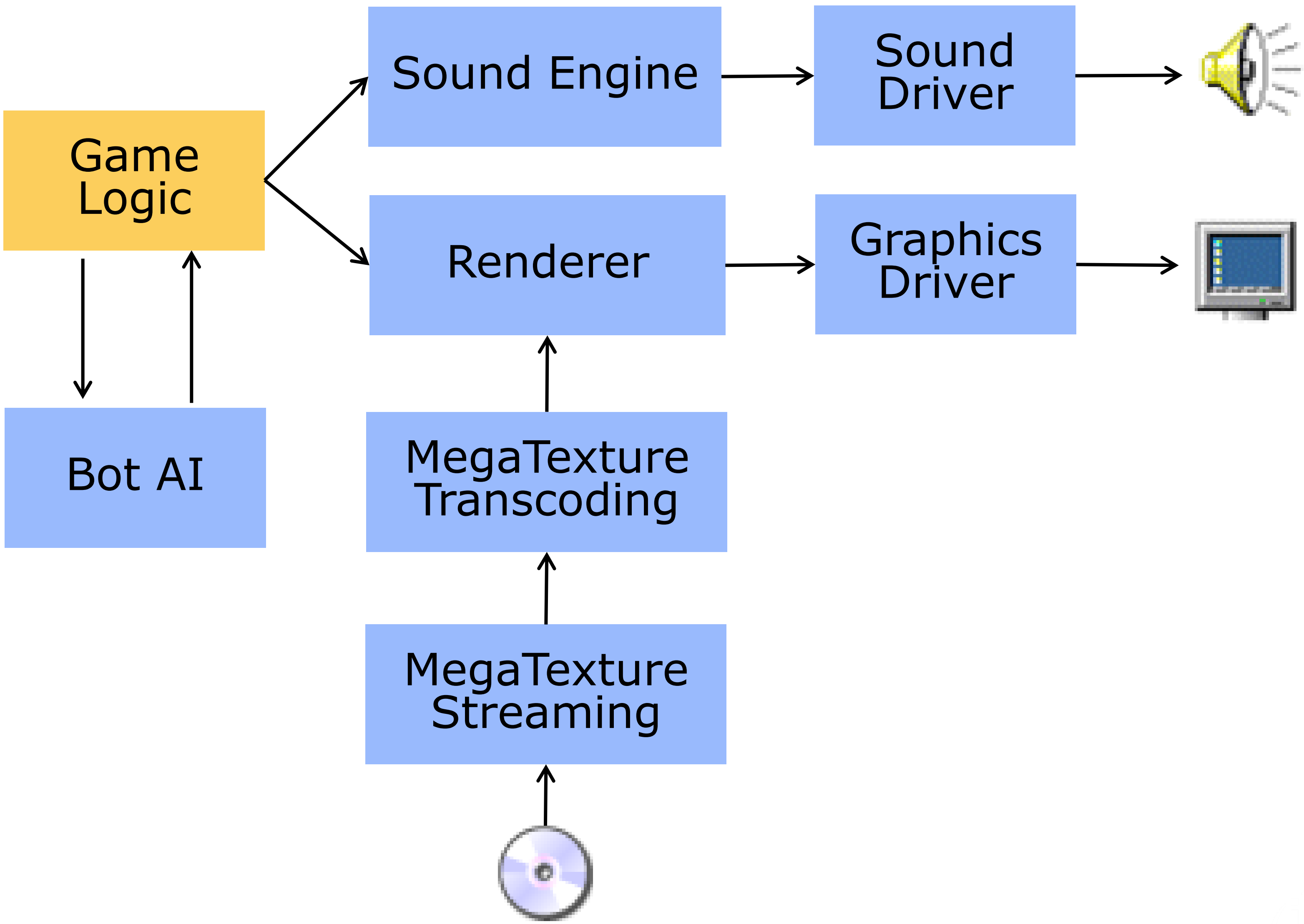
Sound Engine



Sound Engine

- The sound system performs spatialization.
- Decompresses OGG sounds in real-time.
- The sound thread does not consume a whole lot of CPU (typically < 5% on a Core 2).

Game Logic

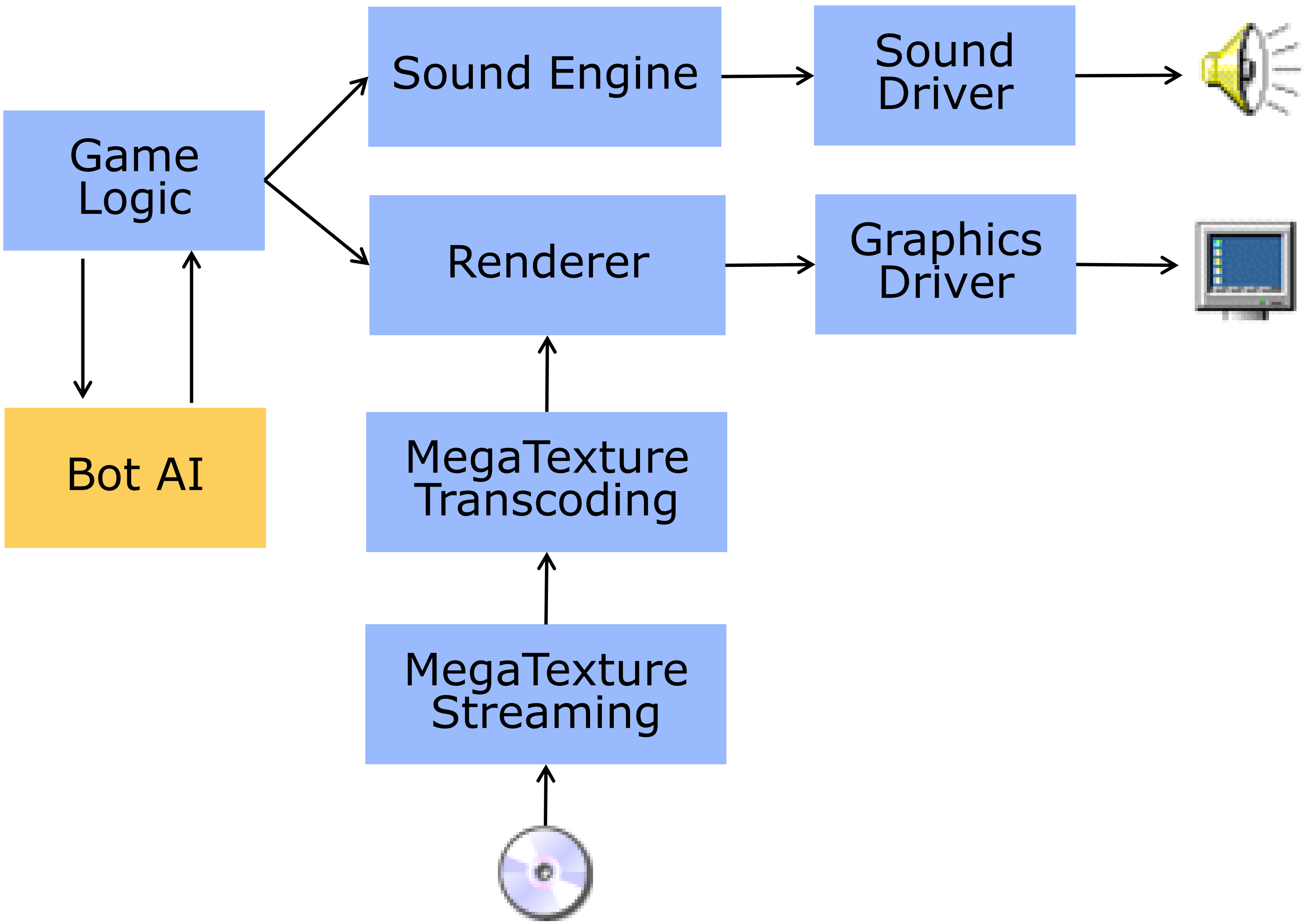


Game Logic
Sound Engine
Sound Driver
Renderer
Graphics Driver
Bot AI
MegaTexture Transcoding
MegaTexture Streaming

Game Logic

- The game logic runs at a fixed 30 Hz.
- The game code consumes quite a bit of CPU.
- A lot of this is collision detection and physics.
- The game logic itself typically involves lots of branchy code and can be expensive as well.

Bot AI



Sound Engine
Sound Driver
Game Logic
Renderer
Graphics Driver
Bot AI
MegaTexture Transcoding
MegaTexture Streaming

Bot AI

- The development of ETQW AI/bots did not start at the beginning of the project.
- On one hand this was a good thing because the AI implements thousands of game dependent rules that would have to change as the game is changed and tweaked during development.
- On the other hand the ETQW AI was developed in about a year which really is a short period of time to develop AI for a game with the complexity of ETQW.

Bot AI

- The AI threading in ETQW was designed and planned from the start.
- As a result the threading had little impact on the development time.
- The threading actually forced us to implement AI with clear data separation from the game code because the data has to be buffered.
- This is a good thing!

Bot AI

- The path and route finding system only run in the AI thread and as such do not need to be "thread safe".
- The collision detection system had to be made thread safe.
- At any point in time the AI can query the current collision state of the world.
- Unfortunately this introduces a source of non-determinism because the AI can query the collision state while the physics, which runs in the game thread, is moving objects around at the same time.

Bot AI

```
static const int MIN_FRAME_DELAY = 0;
static const int MAX_FRAME_DELAY = 4;
HANDLE      gameSignal;
HANDLE      aiSignal;
Int         gameFrameNum;
int         lastAIGameFrameNum;

void GameThread() {
   for ( ; ; ) {
      SetCurrentGameOutputState();
      AdvanceWorld();
      SetCurrentGameWorldState();

      gameFrameNum++
      // let the AI thread know there's another game frame
      ::SetEvent( gameSignal );
      // wait if the AI thread is falling too far behind
      while( lastAIGameFrameNum < gameFrameNum - MAX_FRAME_DELAY ) {
          ::SignalObjectAndWait( gameSignal, aiSignal, INFINITE, FALSE );
      }
   }
}
```

Bot AI

```
void AIThread() {
   for ( ; ; ) {
      // let the game thread know another AI frame has started
      ::SetEvent( aiSignal );
      // never run more AI frames than game frames
      while( lastAIGameFrameNum >= gameFrameNum - MIN_FRAME_DELAY ) {
         ::SignalObjectAndWait( aiSignal, gameSignal, INFINITE, FALSE );
      }
      lastAIGameFrameNum = gameFrameNum;

      SetCurrentAIWorldState();
      AdvanceAI();
      SetCurrentAIOutputState();
   }
}
```

Bot AI

- The last optimization we did in ETQW cut AI CPU usage in half and it took less than a minute to implement. We simply changed the MIN_FRAME_DELAY from zero to one.
- This reduces the think frequency of the AI to 15Hz.
- In Quake III Arena the bots were only thinking at 10Hz.

Threading On/Off

- Always implement an option to switch between threaded mode and non-threaded in real-time.
- This is very useful to see the true performance difference.
- Also makes it much easier when debugging the threaded code.

Common Issues

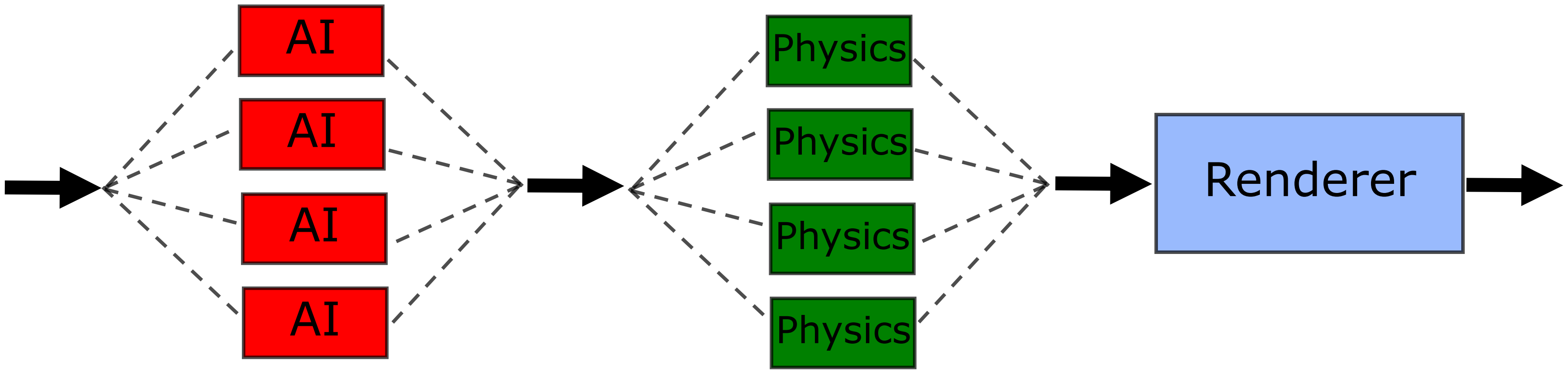
- Load Imbalance
- Under utilization of processors
  - Gustafson’s law increasing the amount of parallel work
  - Adding new features in games like fracture, smoke, cloth, procedural texture
- Amdahl’s law - Need to reduce Serial time to improve scaling
  - Parallelize code as far as possible
  - Vectorize serial code
  - Reduce time spent in a serial memory allocator
- Over subscription
  - Different Threaded subsystems
  - Threading at various levels of the application stack
  - Threaded middleware

Scalability

- PCs have a broad range of capabilities from CPU to Graphics
- Even with a fixed target platform its hard to load balance for real game play.
- Scene complexity, interactivity, physics vary from scene to scene
- Need to think how to make best use of resources
- Granularity Vs Load Balancing
- Common threading infrastructure with priorities/QoS.

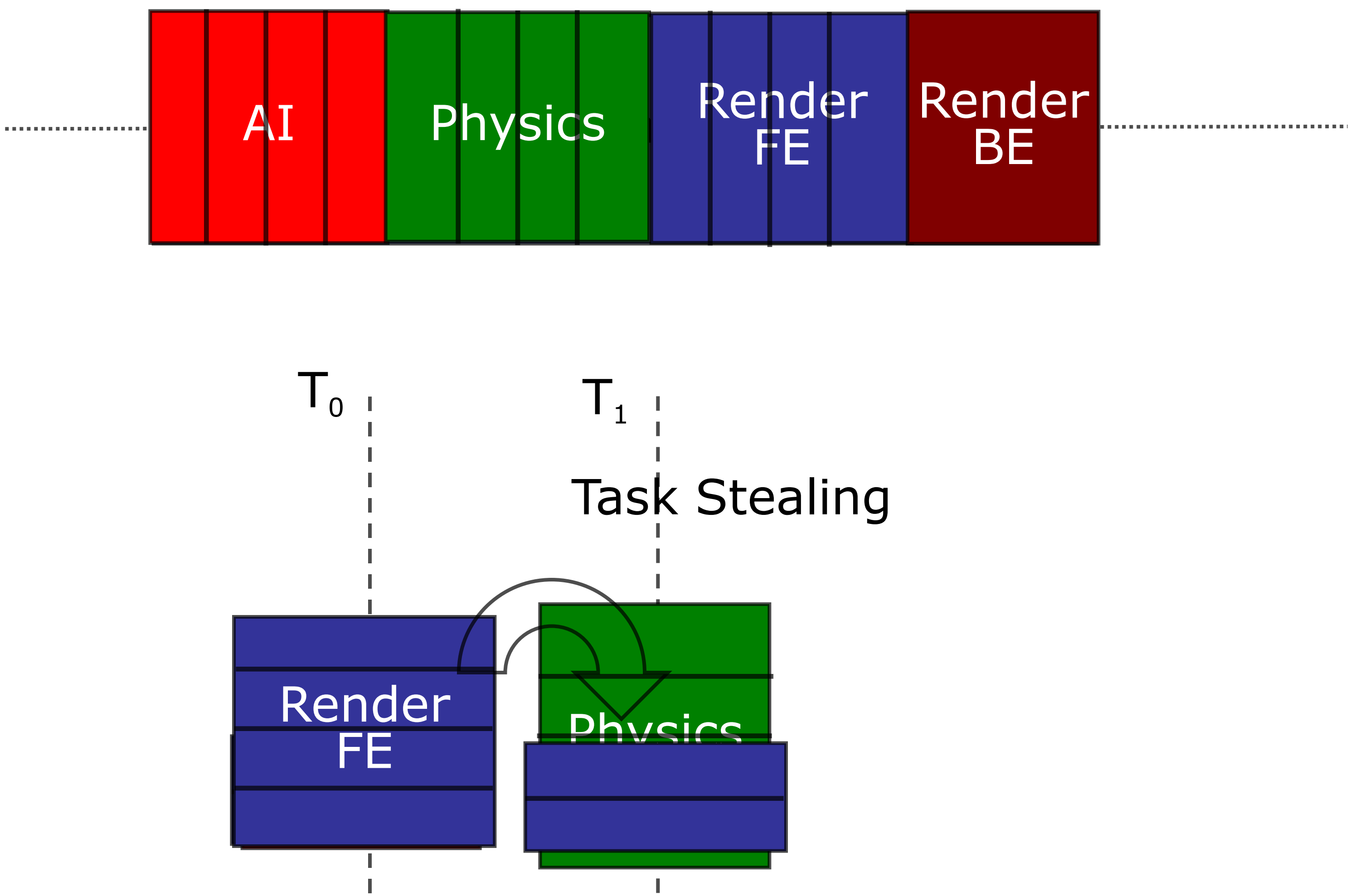
Alternate Threading Paradigms

- Data Decomposition



Alternate Threading Paradigms

- Task/Work decomposition / Pipeline



Design with threading in mind

- Lot easier to thread code that’s designed well.
- Reduce the coupling (data-dependence) between subsystems
- Make them as asynchronous as far as possible.
- Factor a given subsystem into data and operations performed on the data (iterators).
- Make sure that data classes don’t store any iterator data and are reentrant.
- Have a mechanism to ensure validity of shared, mutable data.
- Intel's Threading Building Blocks (TBB) has some good resources like thread safe containers, efficient memory allocator, generic parallel algorithms (parallel for, ….) and its open source.

Summary

- Threading Game Engines is not a trivial task - Game engines are very complex pieces of code with a relatively short shelf life.
- Game engines naturally lend themselves to functional decomposition but interdependence between the various subsystems can cause excessive synchronization and performance overheads.
- Functional decomposition leads to load imbalance and often performance is limited by the main thread. Need to Investigate alternate paradigms like Task Queues to improve load balance.
- Need to design and implement debugging aids into the threading infrastructure
  - Interaction with the GPU makes debugging harder

www.intel.com/software/graphics

Wednesday

10:30am - Gaming on the Go
12:00pm - COLLADA in the Game
02:30pm - Interactive Ray Tracing in Games
04:00pm - Speed Up Synchronization Locks

Thursday

09:00am - The Future of Programming for Multi-Core with the Intel Compilers
10:30am - Getting the Most Out of Intel Graphics
12:00pm - Comparative Analysis of Game Parallelization
02:30pm - Threading Quake 4 and Quake Wars