Study Guide for Game Programming

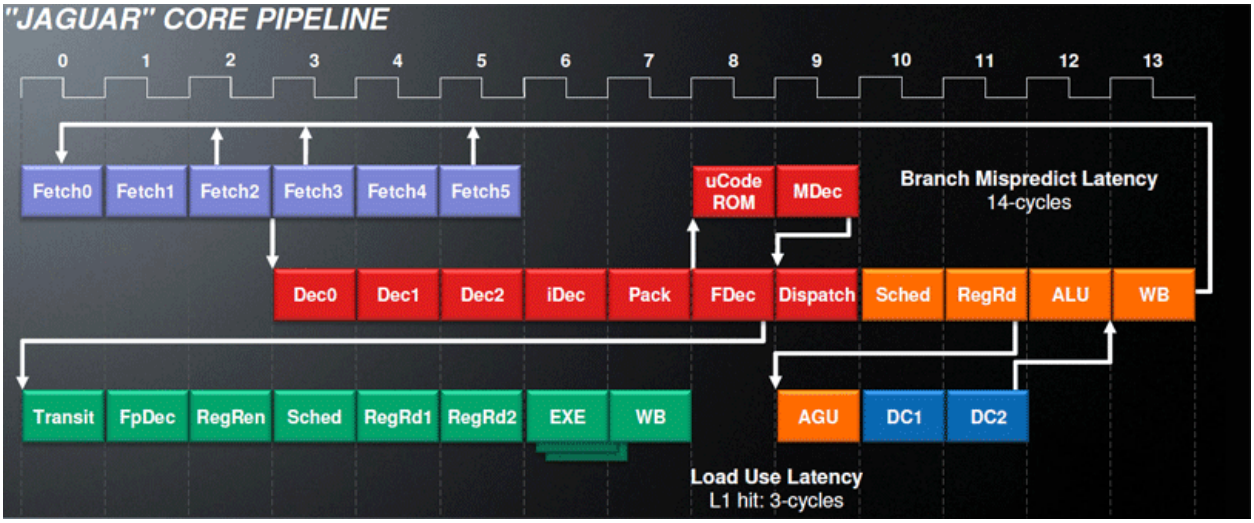
Kareem Omar

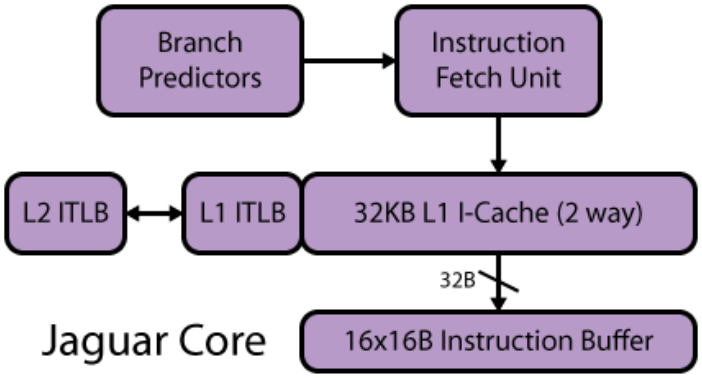
kareem_omar@naughtydog.com

Kareem Omar
kareem_omar@naughtydog.com

Microarchitecture

AMD Jaguar microarchitecture:

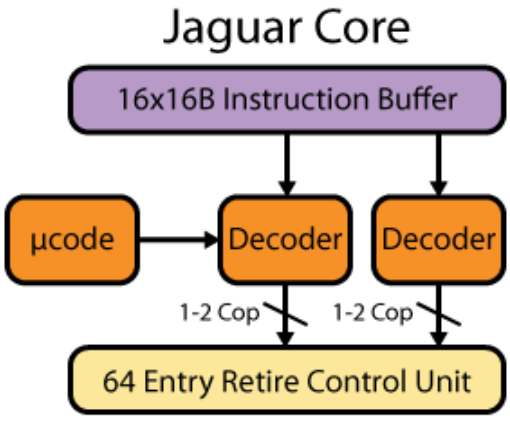
- Family 16h, successor to Bobcat
- 28 nm
- 

- Issues 2 instructions and dispatches 6 operations per cycle
- A cluster of four Jaguar cores share an L2 cache
- The instruction fetch pipeline in Jaguar and Bobcat is a total of six stages. However only the first three stages are on the critical path – the latter three stages occur in parallel with instruction decoding and are primarily used to check that the fetch address was correct and resteer the pipeline if necessary (e.g., to recover from a branch misprediction).
- The L1 instruction cache (L1I) is a 32KB, 2 way associative design with 64B cache lines. The instructions are parity protected, with a pseudo-Least Recently Used (LRU) replacement algorithm. The L1 Instruction Translation Look-aside Buffer (ITLB) is fully associative with 32 entries for 4KB pages and 8 entries for 2MB pages. The larger 1GB pages are fragmented into multiple 2MB pages, since the target markets are relatively unlikely to use huge pages.
- Instruction fetching occurs in 32B fetch windows, which are the basis for much of the front-end, so fetching a full cache line takes at least two cycles.
- 

- When a branch is detected, the IP address of the fetch window indexes into the Branch Target Buffer (BTB), which is coupled to the L1I. The BTB is a two level structure; the L1 is optimized for sparse branches and the L2 handles dense branches. The L1 BTB is conceptually part of the instruction cache; it tracks two branches for every 64B line (1024 entries total) and can simultaneously predict both branches with only a single cycle penalty for taken branches. The L2 BTB is allocated dynamically and tracks an extra 2 branches per 8B region and also contains 1024 entries. The L2 BTB is slower and makes a single prediction per cycle, with a two cycle penalty for the first dense branch prediction and only a single cycle for any subsequent prediction. The BTB design saves power by only engaging the L2 when code actually has 3 or more branches per cache line, exploiting branch density to reduce power.
- Conditional near branches are implicitly predicted as not-taken, which saves space in the BTB. Once such a branch is taken, it is set to always taken in the BTB. Should the always taken branch subsequently fall through, it switches to a dynamic neural network predictor using 26-bits of global history. The Bobcat and Jaguar branch predictor proved to be so successful that it was later adopted for AMD's big cores, particularly Piledriver.
- Another BTB optimization is that the L1 and L2 BTBs only predict target addresses for direct branches that are in the same 4KB page as the IP of the fetch window. A 32-entry out-of-page target array handles branch targets with up to 256MB of displacement for the L1 BTB. Sparse branch targets with >256MB of displacement, and dense branches with out-of-page targets are resolved by the branch target address calculator with a four cycle penalty.
- Near calls and the associated returns are predicted by a 16 entry Return Address Stack (RAS). The RAS can recovery from most forms of misspeculation without corrupting the predictions. For cases that cannot be recovered, the RAS is invalidated to avoid mispredictions.
- Indirect branches with multiple targets are predicted using the IP address and 26-bits of global history to index into the 512-entry indirect branch target array. There is an extra 3 cycle penalty for any indirect branch predictions, but indirect branches with a single target and 256MB or less displacement are tracked through the lower latency out-of-target array.
- If a cache line is only being used for instructions, then the branch information in the L1 BTB is compressed and stored in the ECC bits of the L2 cache when the line is evicted and can be reloaded. The information is lost if the cache line is hit by a store, or is evicted to main memory. L1I misses trigger a 64B fetch request to the L2, and also prefetch one or two additional cache lines.
- Once the fetch address has been determined, the 32B of instructions from the L1I are sent to the Instruction Byte Buffer (IBB), which acts as a decoupling queue between the fetch and decoding stages. The IBB entries are 16B each, so a fetch will typically fill two at a time, and

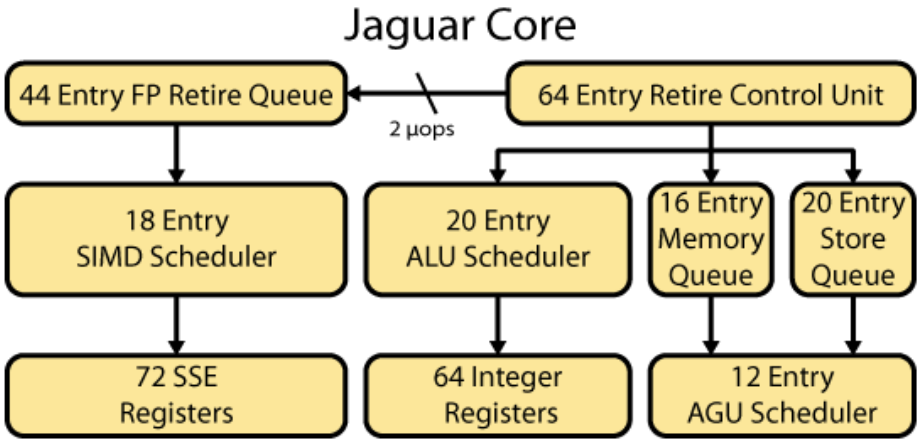
Jaguar has 16 entries. A small loop buffer tracks four recent 32B fetches and can bypass the instruction cache lookup mechanism to save power.

- Jaguar can decode two x86 instructions per cycle. Instructions are decoded into fixed length but relatively complex micro-operations (COPs) that are tracked by the control logic. The actual COPs can map to multiple µops executed by the hardware. A single COP is capable of performing a read from memory, an ALU operation, and a write to memory, but most COPs map to one or two µops.
- Instruction decoding in Jaguar starts in the fourth cycle and overlaps with the latter part of the fetch pipeline.

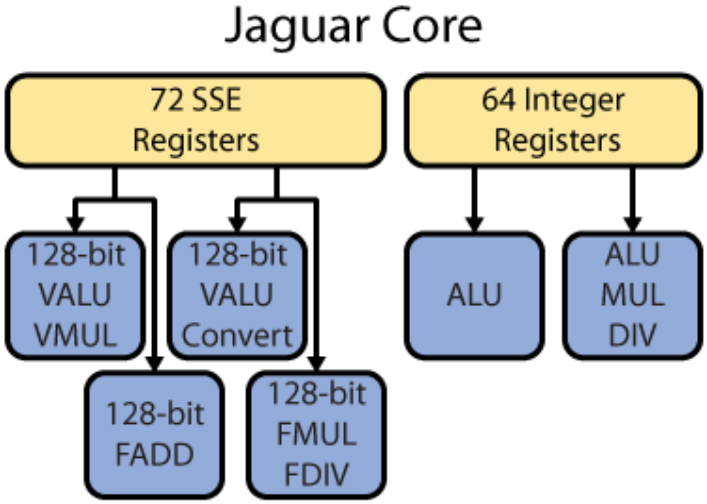


- Jaguar has 128-bit vector units (but supports AVX).
- Some of the SSE4.2 string instructions and cross-lane AVX instructions are microcoded in Jaguar.
- The decoding stages for Jaguar also incorporates a side-band stack optimizer. Instructions such as CALL, RET, PUSH, and POP implicitly modify the stack pointer, creating long dependency chains. The stack optimizer splits the stack pointer into two dedicated registers, a base register and a delta register. The stack pointer manipulations are all performed on the delta register by a dedicated ALU in the front-end and the two registers are synchronized as needed. This eliminates µops from the out-of-order machine and improves IPC.
- The last decoding pipeline stage for Jaguar packs two COPs, assigning them to lanes based on execution unit restrictions and writes them into an instruction queue. The instruction queue can hold ~6-8 entries and absorbs any delays due to stalls later in the pipeline.

- Once instructions have been decoded into COPs, they are dispatched to the back-end of the CPU core for renaming, scheduling, and issuing to the execution units. The FP and integer clusters are renamed separately, and the schedulers are distributed.
- The Jaguar back-end can dispatch two COPs per cycle from the instruction queue. The first cycle of dispatching (FDEC) checks to ensure that the necessary downstream resources are available, and stalls COPs as necessary. The next pipeline stage, dispatch, performs the renaming and the scheduling stage writes into the scheduling queues.

- 

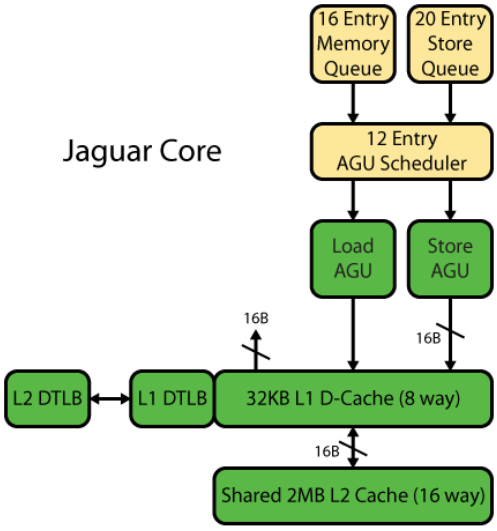
- In Jaguar, AVX instructions that operate on 256-bit data are cracked into two COPs; register-register operations will decode into a single µop, whereas register-memory operations spawn two µops. So in theory, an instruction such as VMULPD can spawn 4 µops – 2x128b loads and 2x128b multiplies.
- To enable issuing two FP/SIMD µops per cycle, the physical register file has 4 read ports and 3 write ports, each port is 128-bits wide. The last write port is used for load instructions that target registers in the FP cluster.

- 

- The basic integer µop data flow is three cycles. Once the µop is written into the scheduler, there may be a variable length delay – driven by the availability of input operands as well as execution resources. When a µop is ready to execute, it reads input operands in the first cycle and then is actually executed in the next cycle. The result is immediately available for forwarding, but the actual register write back takes a third and final cycle.
- The integer scheduler can issue two µops per cycle. The two ALU pipelines are symmetrical and fully pipelined with single cycle throughput for nearly all basic operations. However,

integer multiplier and divider hardware is relatively expensive and is only available on ALU1. 32b integer multiplication is fully pipelined with 3 cycle latency, while 64b integer multiplies have one quarter throughput and 6 cycle latency. The divider is derived from Llano and is an unpipelined, radix-4 design that computes 2 bits of the result each cycle. Additionally, 3-operand LEA instructions use the store AGU in the first cycle and feed into ALU1 for the second.

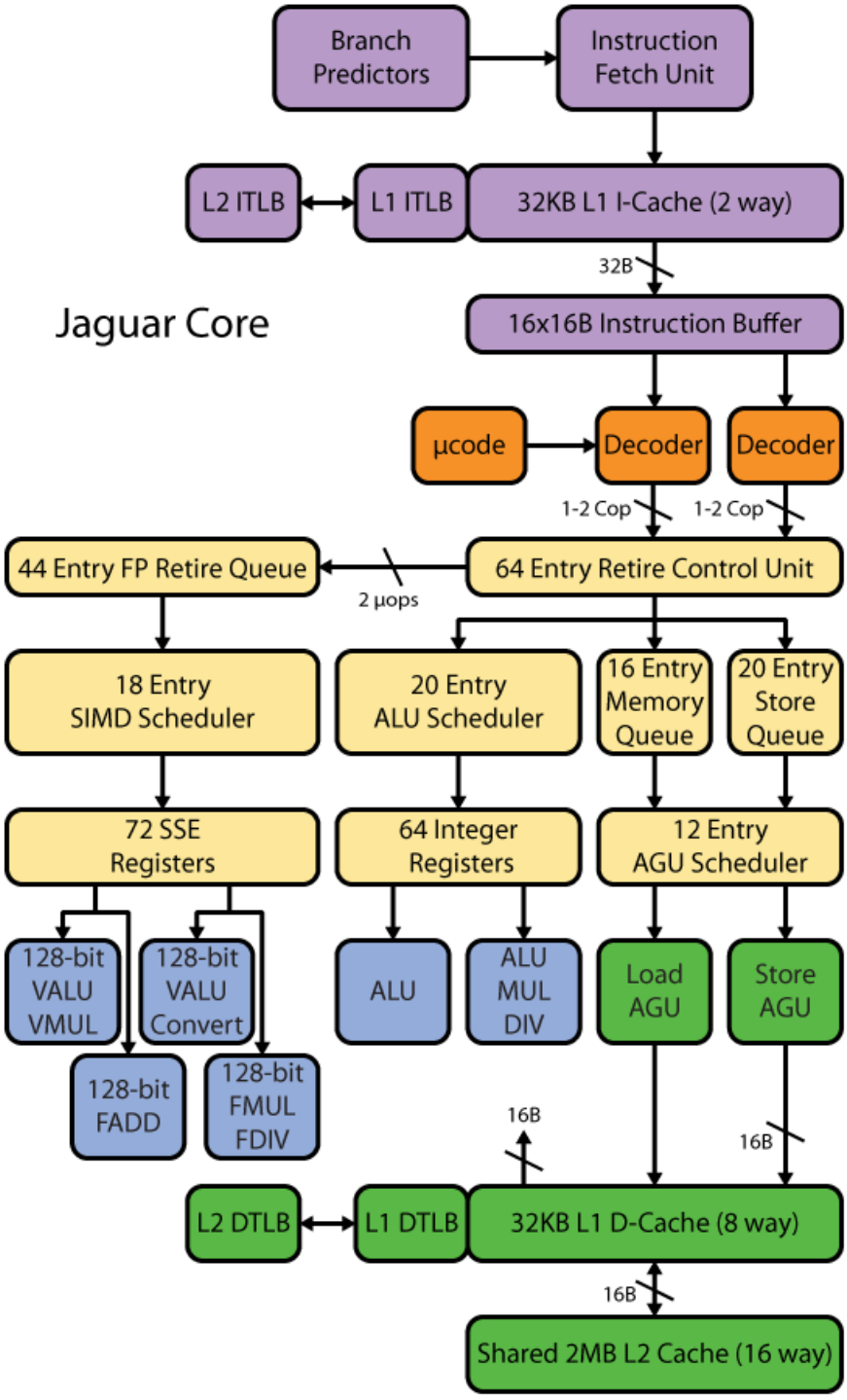
- On Jaguar, port 0 contains a fully pipelined 128-bit SIMD integer multiplier that handles IMUL operations in two cycles. The integer multiply is also used for AES and carry-less multiplication, although these instructions are microcoded.
- The store and conversion unit hangs off port 1 and handles FP/integer data conversion, most floating point denormals, as well as routing up to 128-bits of write data to the L1D. The latency is 3 cycles for most operations, but fully pipelined.
- The asymmetry is more pronounced on the floating point side. On Jaguar, port 0 is equipped with a 128-bit FP adder that is pipelined for single and double precision, with 3 cycle latency. The adder also handles denormals for SSE and AVX operations (but not x87).
- FP multiplication and division is resolved on port 1. Jaguar's FP multiplier is implemented as two 76-bit × 27-bit multipliers to save area and power. For single precision data, it is fully pipelined and executes with 2 cycle latency for up to 4 operations. However, double precision operations require an additional iteration – doubling the latency and cutting throughput in half. Rarer 80-bit x87 multiplication takes 5 cycles and can only execute every third cycle.
- Port 1 also executes FP division and square roots, re-using the multiplier circuitry and therefore blocking any multiply µops. Division is only available through the deprecated x87 instruction set and takes 14-22 cycles depending on the specific variant. Exact square root instructions similarly have a latency of 16-35 cycles, but certain reciprocal square root instructions (e.g., RSQRTPS, RSQRTSS) are fully pipelined with 2 cycle latency.
- At dispatch, any µop accessing memory starts by allocating an entry in the AGU scheduler and the memory ordering queue (MOQ); store µops also reserve an entry in the store queue.

- 

- The AGU in Jaguar is optimized for simpler addressing and impose a one cycle penalty if either the segment or base registers are non-zero, suggesting that the AGU contains a single adder and multiplier.
- Once the virtual address has been calculated, it is written into the memory ordering queue (MOQ). Memory µops then access the data cache and check the store queue for memory ordering constraints. The L1 data cache is 32KB and 8-way associative with 64B cache lines, so that the physical and virtual address only differ in the tags (rather than the index). The writeback cache is parity protected with a pseudo-LRU replacement algorithm. Jaguar's L1D can sustain a 128-bit read and a 128-bit write each cycle, doubling the bandwidth of Bobcat and also avoiding certain bank conflict scenarios.
- The actual data cache access takes two clock cycles. In the first cycle, the virtual address is translated in parallel with checking the data cache tags for a hit. The L1 DTLB is fully associative and holds 40 entries for 4KB pages and 8 entries for larger 2MB pages. While 1GB pages are supported, they are fractured into 2MB pages. The L2 DTLB includes a 512 entry, 4-way associative array for 4KB pages and a 256 entry, 2-way associative array for 2MB pages. L2 TLB misses are speculatively resolved by a hardware page table walker that includes a 16-entry page directory cache for intermediate translations. Stores exit the MOQ once the physical address is available and written into the store queue.
- If the tag check determines that a hit has occurred in any of the 8 ways of the L1D, the second cycle will be used to read out the data and format it appropriately. Once data from a load has been delivered to registers or the forwarding network, the load µop has completed and may exit the MOQ and wait in a separate queue till retirement. The load to use latency is 3 cycles, but the bypass network adds two extra cycles of latency for transmitting data to the FP cluster. Misaligned loads within a 16B boundary generate a single cache access, whereas those crossing a 16B boundary suffer at least one extra cycle of latency and half

throughput. Unaligned load instructions (e.g., MOVUPD) to aligned data suffer no latency penalties.

- A group of four cores shares a single L2 cache control block that controls 4 banks of L2 data arrays and acts as the central point of coherency and interface to the rest of the system. For systems with more than four cores, each local cluster must be connected through a fabric.
- The Jaguar shared L2 cache is 1-2MB and 16-way associative with ECC protection for data and tag arrays. The minimum hit latency is 25 cycles. The L2 is inclusive of all 8 lower level caches (4 data caches and 4 instruction caches), thereby acting as a snoop filter and has a write back policy. The L2 cache is implemented with a single shared interface block and 4 replicated data arrays.
- The L2 cache interface operates at core frequency and includes the control logic, tag arrays, and bus interfaces to the cores and rest of the system. The L2 controller can track 24 transactions, each pairing a read and a write. The L2 incorporates a prefetcher for each core and a 16 entry snoop queue for coherency requests. The L2 tags are split into 4 banks with addresses hashed across the banks. Each tag array is associated with a data array, so an L2 tag hit will only check a single bank of the data array for the cache line. This hashing technique reduces the clocking of the tag and data arrays, thereby decreasing power consumption. The data arrays are 512KB each, and run at half core frequency to save power. Each data array delivers 16B per core clock cycle, for an aggregate bandwidth of 64B/cycle.



-
- Any repetitive pattern with a period of n+1 or less can be predicted perfectly after a warm-up time no longer than three periods. A repetitive pattern with a period p higher than n+1 and less than or equal to 2 n can be predicted perfectly if all the p n-bit subsequences are different.

Agner Fog's notes:

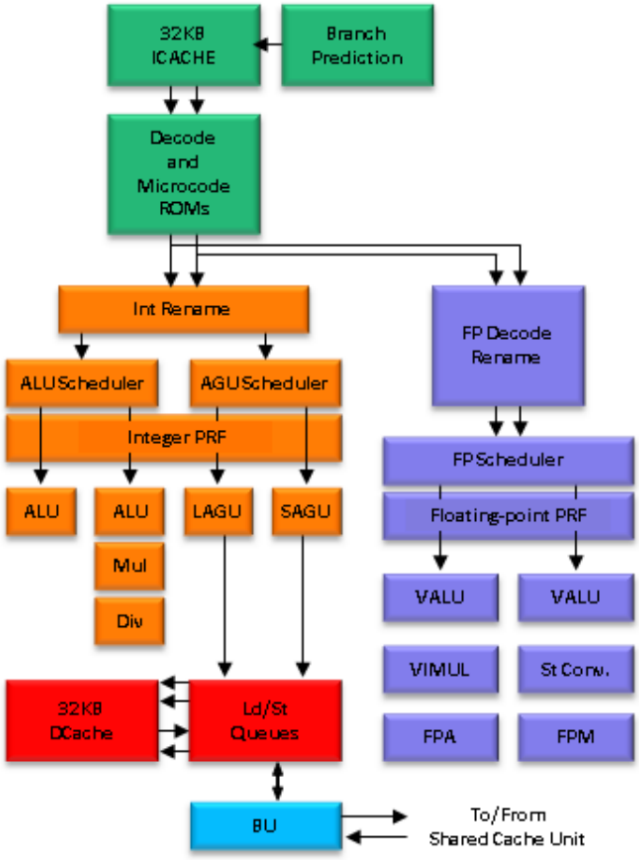
- the position of jumps and branches is stored in two arrays, a "sparse branch marker array" which can hold 2 branches for each 64-bytes cache line, and a "dense branch marker array" which can hold 2 branches for every 8 bytes of code, according to the article cited below. In my tests, the Bobcat could predict 2 branches per 64 bytes of code in the level-2 cache, which indicates that the sparse branch array, but not the dense branch array, is coupled to both levels of cache. If both arrays are used, we would expect a maximum of 18 branches
- 35 per 64 bytes of level-1 cache. In my tests, the Bobcat and Jaguar were able to predict 16 or
- 17 branches per line of level-1 cache, depending on the position of the branches, but not
- 18. There are many mispredictions when these number of branches are exceeded, as the
- branches keep evicting each other from the arrays. Branches that are never taken are included in the branch marker arrays. Misprediction penalty The article cited below indicates a misprediction penalty of 14 clock cycles. In my tests, however, the misprediction penalty ranged from 8 to 19 clock cycles depending on the subsequent instructions. There was no consistent difference between integer and floating point code in this respect.
- Pattern recognition for conditional jumps The branch predictor behaves approximately as a two-level adaptive branch predictor with a global 26-bit history (see page 15), or slightly better. There is no dedicated loop predictor. This means that loops with many branches or other loops inside are poorly predicted. I have no information on the size of the pattern history table, but we can surely assume that it is less than 2^26. Branches that are always taken are included in the history buffer, while unconditional jumps and branches never taken are not. Prediction of indirect branches The processor can predict indirect jumps with more than two different targets, but mispredictions are frequent. Return stack buffer: there is a return stack buffer with 16 entries on Jaguar according to my measurements.
- Taken jumps have a throughput of one jump per two clock cycles. It is delayed another clock cycle if there is a 16-byte boundary shortly after the jump target. Not taken branches have a throughput of three per clock cycle. Avoid a one-byte return instruction immediately after a branch instruction.
- The pipeline is designed for a throughput of 2 instructions or µops per clock cycle with two integer ALUs, two floating point/vector units, and two AGUs for load and store, respectively.
- The pipeline has schedulers for out-of-order execution. There are two separate pairs of pipelines, one for general purpose integer instructions and one for floating point and vector instructions. The general purpose integer unit has the two integer ALUs, the load unit and the store unit. The floating point and vector unit has two pipes.
- Actual instruction fetch rate appears to be 16 bytes per clock, not 32.
- No loop buffer.
- Decodes 2 instructions per cycle.
- No prefix penalties at all.
- No macro-fusion of branches.
- Lots of stuff is 1 µop. Calls and most AVX are 2 µops. Anything more than 2 µops is microcoded.

- There are two execution pipelines for integer µops with two almost identical ALUs. A few integer instructions, including multiplication, division, and some versions of LEA, can only be handled by integer pipe 0, other instructions can be handled by either pipe. Integer division uses the floating point division circuit with a large extra delay on Bobcat. Integer division is much faster on Jaguar.
- There is one load unit and one store unit, 128 bits wide each on Jaguar.
- There are two floating point execution pipelines, which are also used for integer vector operations. Both pipelines can handle move, integer addition, and Boolean operations.
- Floating point pipe 0, here called FP0, handles integer multiplication and floating point addition. FP1 handles floating point multiplication and division. The division circuit is not pipelined and has no early out feature, hence the latency for division is independent of the values of the operands.
- Memory read and write operations use the integer load/store unit with an extra delay of one or more clock cycles.
- The Jaguar has penalties of 40 clock cycles for addition of subnormal numbers and 170 - 210 clock cycles when a multiplication gives a subnormal result or underflow.
- Zero register but no other move elimination.
- 1 cycle delay for transport from integer to FP vector or vice versa.
- 3 cycle penalty for memory read or write that crosses a 16-byte boundary, plus halved throughput since you now need two reads/writes.
- Store forwarding is pretty fast, particularly for 32- and 64-bit GPRs at 3 cycles – that's faster than Intel! It's 7 for XMM.
- PREFETCH is super slow. Don't use it.
- Denormal numbers, NAN's, infinity and exceptions all increase the delays.

AMD Wisdom:

- The AMD Family 16h processor implements the AMD64 instruction set by means of COPs (what the AMD manual calls macro-ops, the primary units of work managed by the processor) and micro-ops (the primitive operations executed in the processor's execution units).

- Most instructions break into one (“fast-path single”) or two (“fast-path double”) macro-ops, which in turn can be any number of micro-ops. Beyond this it’s microcoded.



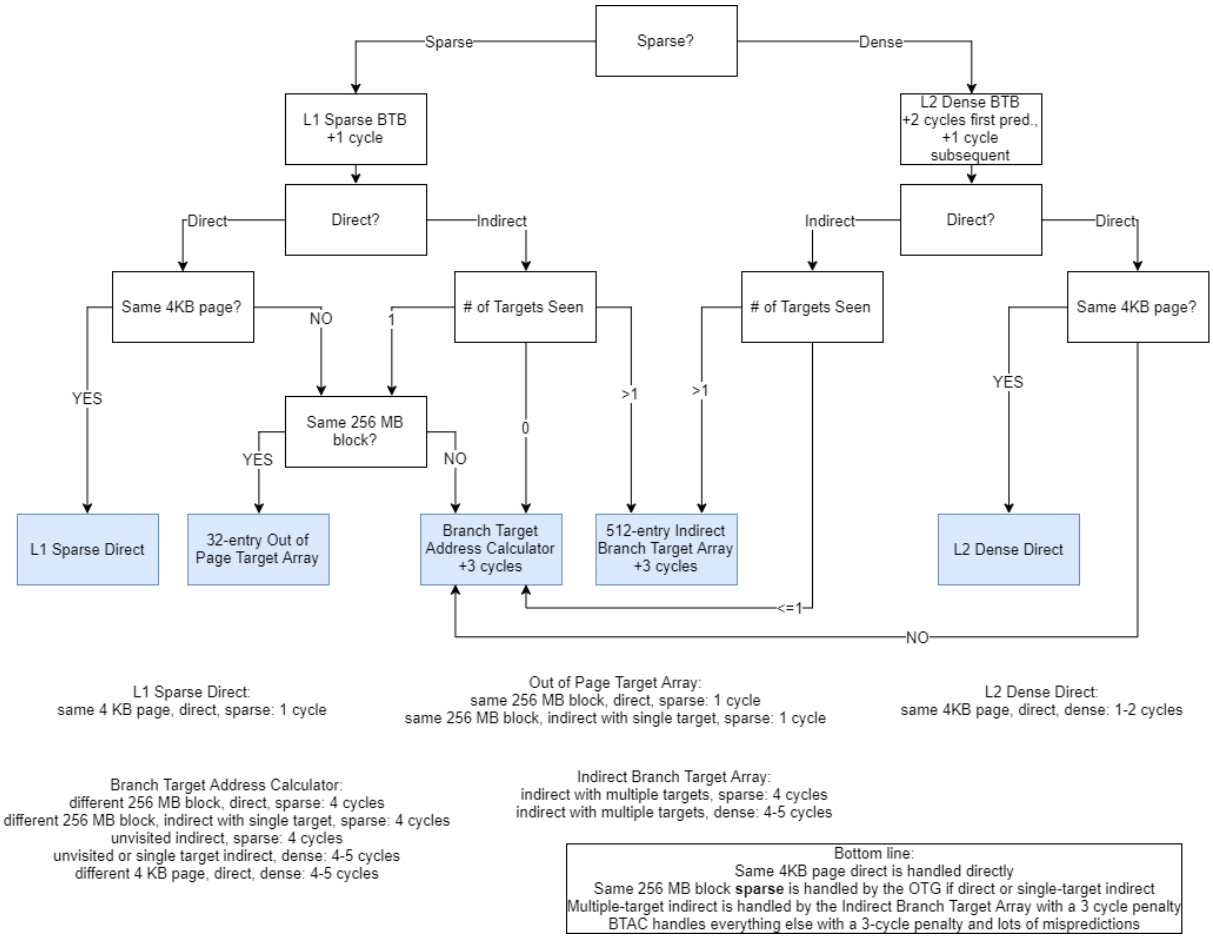
Abbreviation	Descriptive Name
Int Rename	Integer Register Rename Unit
Floating-point PRF	Floating-point Physical Register File
Integer PRF	Integer Physical Register File
FP Decode Rename	Floating-point Decode / Rename Unit
ALU	Arithmetic and Logic Unit
LAGU	Load Address Generation Unit
SAGU	Store Address Generation Unit
BU	Bus Unit
VALU	Vector ALU
VIMUL	Vector Integer Multiply Unit
St Conv.	Store / Convert Unit
FPA	Floating-point Addition Unit
FPM	Floating-point Multiplication Unit
Ld/St Queues	Load / Store Queues

Diagram Legend

- The L1 data cache has a 3-cycle integer load-to-use latency, and a 5-cycle FPU load-to-use latency.
- For aligned memory accesses, the aligned and unaligned load and store instructions (for example, MOVUPS/MOVAPS) provide identical performance.
- Natural alignment for both 128-bit and 256-bit vectors is 16 bytes. There is no advantage in aligning 256-bit vectors to a 32-byte boundary on the Family 16h processor because 256-bit vectors are loaded and stored as two 128-bit halves.
- The L2 cache has a variable load-to-use latency of no less than 25 cycles.
- The L2 has four 512-Kbyte banks. Bits 7:6 of the cache line address determine which bank holds the cache line. For a large contiguous block of data, this organization will naturally spread the cache lines out over all 4 banks. The banks can operate on requests in parallel and can each deliver 16 bytes per cycle, for a total peak read bandwidth of 64 bytes per cycle for the L2. Bandwidth to any individual core is 16 bytes per cycle peak, so with four cores, the four banks can each deliver 16 bytes of data to each core simultaneously. The banking scheme provides bandwidth for all four cores in the processing complex that can achieve the level that a private per-core L2 would provide.

- The branch target buffer (BTB) is a two-level structure accessed using the fetch address of the current fetch block. Each BTB entry includes information about a branch and its target. The L1 BTB is a sparse branch predictor and maps up to the first two branches per instruction cache line (64 bytes), for a total of 1024 entries. The two branches in the sparse predictor can be predicted in the same cycle. The L2 BTB is a dense branch predictor and contains 1024 branch entries, mapped as up to an additional 2 branches per 8 byte instruction chunk, if located in the same 64-byte aligned block. Predicted-taken branches incur a 1-cycle bubble in the branch prediction pipeline when they are predicted by the L1 BTB (sparse predictor). The L2 BTB (dense predictor) can predict one additional branch per cycle, with the first dense branch prediction incurring a 2-cycle bubble and subsequent predictions incurring one additional cycle per branch. Predicting long strings of branches in the same cache line through the dense predictor only occurs as long as the branches are predicted not-taken, as the string will be broken by a predicted taken branch. An additional 3-cycle latency is incurred for branch targets predicted through the indirect target predictor or fixed up with the branch target address calculator. For example, an indirect branch predicted by the sparse predictor will incur a 4-cycle bubble. The minimum branch misprediction penalty is 14 cycles. The dense branch predictor can predict one additional branch per cycle, with the first dense branch prediction incurring a 2-cycle bubble and subsequent predictions incurring one additional cycle per branch. Predicting long strings of branches per line through the dense predictor only occurs as long as the branches are fall-through, as the string will be broken by a predicted taken branch.
- The out-of-page target array (OPG) holds the high address bits ([28:12]) for 32 targets that are outside the current page for branches marked in the sparse BTB. Only sparse branches are eligible for out-of-page target prediction. Branches marked by the dense predictor are not eligible for OPG target prediction. Direct dense branches that are out-of-page will have their targets corrected by the branch target address calculator with a 4-cycle penalty. Direct sparse branch targets that cross a 28-bit address block boundary (beyond the range of the out-of-page target array) are also corrected by the branch target address calculator.

Kareem Omar
kareem_omar@naughtydog.com



-
- The Family 16h processor includes a loop buffer which can reduce power consumption when hot loops fit entirely within it. The loop buffer is composed of four 32-byte chunks and is essentially a subset of the instruction cache. Two of the 32-byte chunks must be in the lower 32 bytes of a 64-byte cache line, and the other two must be from the upper 32 bytes of a cache line. Hot code loops that can fit within the loop buffer can save power by not requiring the full instruction cache lookup. Compilers may choose to align known-hot loops to a 32-byte boundary if doing so ensures that they fit completely within the loop buffer. The loop buffer is a power optimization, not an instruction throughput optimization, although in a system with Bidirectional Application Power Management or performance boost enabled, this feature may allow sufficient power budget savings to enable boosting the clock rate to a higher frequency.
- One STLF pitfall to avoid is aliases where store/load virtual address bits [15:4] match, but mismatch in the range [47:16] because it can delay execution of the load. (i.e. 64k apart false dependency)

Kareem Omar
kareem_omar@naughtydog.com

Abbreviated look at Skylake (assumes familiarity with all prior Intel microarchs)

The reorder buffer has 224 entries on Skylake. The reservation station has 97 entries. The Skylake has 180 integer registers and 168 vector registers, according to Intel publications.

The instruction fetch unit can fetch a maximum of 16 bytes of code per clock cycle in single threaded applications. There are four decoders, which can handle four instructions (five with fusion) generating up to four µops per clock cycle in the way described on page 122 for Sandy Bridge. Instructions with any number of prefixes are decoded in a single clock cycle. There is no penalty for redundant prefixes. The penalty for length-changing prefixes is the same as for Sandy Bridge (see page 122). Arithmetic and logic instructions with an immediate operand using an operand size prefix, e.g. add ax,1234 has a penalty of 2-3 clock cycles in the decoders, regardless of alignment. This applies to all arithmetic and logic instructions with a 16-bit immediate constant as operand in 32-bit or 64-bit mode. Move instructions have no penalty for length-changing prefixes.

The loop buffer is used for tiny loops of up to 30 - 40 instructions. The throughput is 4 µops per clock cycle with no restriction on instruction length. The µop cache is used for loops up to approximately 1000 instructions. The throughput is up to 4 instructions or 32 bytes of code per clock cycle. The fetch and decode units are used for instructions that are not in the µop cache. The throughput is up to 4 instructions or 16 bytes of code per clock cycle.

Upper halves of YMM registers are off to save power for 14 us warmup. They shut down again after 675 us of no use. Any instruction with YMM registers will start the warmup process, except vzeroupper and vzeroall. The processor has a penalty of approximately 129 clock cycles in all cases where an operation on normal numbers gives a subnormal result. There is a similar penalty for a multiplication or division between a normal and a subnormal number, regardless of whether the result is normal or subnormal. There are no other penalties.

Kareem Omar
kareem_omar@naughtydog.com

Math

- Triangle strips vs. triangle fan:

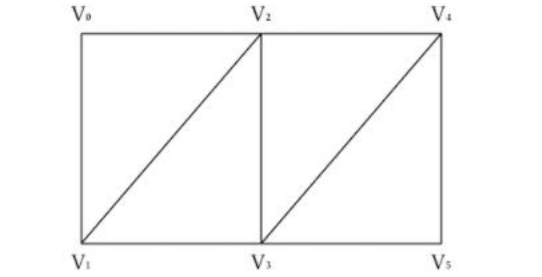


Figure 10.8. A triangle strip.

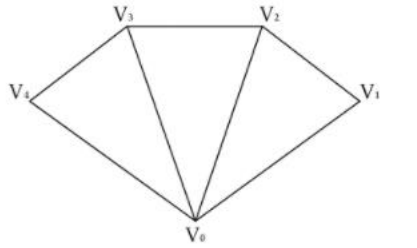


Figure 10.9. A triangle fan.

-

Cauchy-Schwarz Inequality: $|P \cdot Q| \leq \|P\|\|Q\|$ since $P \cdot Q = \|P\|\|Q\| \cos(\alpha)$ and $|\cos(\alpha)| \leq 1$

Law of Cosines:

$$\|P - Q\|^2 = \|P\|^2 + \|Q\|^2 - 2\|P\|\|Q\| \cos(\alpha) = \boxed{\|P\|^2 + \|Q\|^2 - 2P \cdot Q}$$

Can also come from the fact that a dot product against itself is the squared magnitude:

$$\|P - Q\|^2 = (P - Q) \cdot (P - Q) = \|P\|^2 + \|Q\|^2 - 2P \cdot Q$$

Similarly for positive difference:

$$\|P + Q\|^2 = (P + Q) \cdot (P + Q) = \|P\|^2 + \|Q\|^2 + 2P \cdot Q$$

Vector projection:

length of projection of vector P onto Q is just

$$\|P\| \cos(\alpha) = \frac{\|P\|\|Q\| \cos(\alpha)}{\|Q\|} = \frac{P \cdot Q}{\|Q\|}$$

And therefore, sticking this magnitude onto the Q direction:

$$proj_Q P = \frac{P \cdot Q}{\|Q\|}\left(\frac{Q}{\|Q\|}\right) = \frac{P \cdot Q}{\|Q\|^2} Q$$

And just subtract that away from the original vector to get the perpendicular component:

$$perp_Q P = P - \frac{P \cdot Q}{\|Q\|^2} Q$$

Cross-product pseudodeterminant:

$$P \times Q = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ P_x & P_y & P_z \\ Q_x & Q_y & Q_z \end{vmatrix}$$

$$(P \times Q) \cdot R = \begin{vmatrix} P_x & P_y & P_z \\ Q_x & Q_y & Q_z \\ R_x & R_y & R_z \end{vmatrix}$$

$$P \times Q = \|P\| \|Q\| \sin(\alpha)\, n$$

$\|P \times Q\|$ gives the area of the parallelogram defined by P and Q. Therefore $\frac{1}{2}\|P \times Q\|$ gives the area of the triangle with two sides P and Q. (Those can be subtractions of vertices as well, so $\frac{1}{2}\|(V_2 - V_1) \times (V_3 - V_1)\|$).

Unit vectors:

$$i \times j = k$$

$$j \times k = i$$

$$k \times i = j$$

$$j \times i = -k$$

$$k \times j = -i$$

$$i \times k = -j$$

$$Q \times P = -(P \times Q)$$
$$(P \times Q) \cdot R = (R \times P) \cdot Q = (Q \times R) \cdot P$$
$$P \times (Q \times P) = P \times Q \times P = P^2 Q - (P \cdot Q) P$$

Cross products are anticommutative and are NOT ASSOCIATIVE (except in the special case above)

Gram-Schmidt Orthogonalization: subtract projection of all previous vectors from current one, normalize

Matrices:
Matrices are not commutative, but they ARE associative.

Linear systems for which the constant vector is zero are called *homogeneous*.

Homogeneous linear systems always have at least one solution – the zero vector. Nontrivial solutions exist only when the reduced form of the coefficient matrix possesses at least one row of zeros.

Determinant:

Exchanging two rows negates the determinant.

Multiply a row by a scalar *a* multiplies the determinant by *a*.

Adding a multiple of one row to another has no effect.

Cofactor matrix C

$$F^{-1} = \frac{C_{ji}(F)}{detF}$$

Inverse of 2x2 is 1/det * $\begin{bmatrix} flip & neg \\ neg & flip \end{bmatrix}$

Cramer's Rule:

$$x_k = \frac{detM_k(r)}{detM}$$

Not efficient for solving, but Cramer's rule does tell us that if the coefficients and constants in a linear system are all integers and det M = ± 1, then the unknowns must all be integers.

Eigenvalues:

$$Mv = \lambda v$$

$$(M - \lambda I)v = 0$$

In order to have nontrivial v, we require:

$$\det(M - \lambda I) = 0$$

Eigenvectors:

$$(M - \lambda_i I)V_i = 0$$

The eigenvalues of a **symmetric** real matrix are real.

Any two eigenvectors associated with **distinct** eigenvalues of a symmetric matrix are **orthogonal**.

Diagonalization:

If we can find *n* linearly independent eigenvectors, *M* can be diagonalized by A where A's columns are the eigenvectors and $A^{-1}MA$ is diagonal and has the eigenvalues as diagonal elements.

Any symmetric matrix can thus be diagonalized, with $D = A^{-1}MA = A^T MA$.

Linear Transformations:

The column vectors of a transformation matrix, u, v, and w, represent the images in C' of the basis vectors of C (i, j, and k).

Orthogonal: a matrix is orthogonal if $M^{-1} = M^T$, in other words if its inverse is equal to its transpose.

If the column vectors for an orthonormal set, the matrix is orthogonal!

Orthogonal matrices preserve length:

$$\|MP\| = \|P\|$$

And angles:

$$(MP_1) \cdot (MP_2) = P_1 \cdot P_2$$

Performing an odd number of reflections reverses handedness. An even number of reflections is equivalent to a single rotation. Any series of reflections can be regarded as a rotation and then either 0 or 1 reflection. -1 determinant means a rotation and reflection (improper). 1 determinant means rotation only (proper).

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Kareem Omar
kareem_omar@naughtydog.com

4D affine transforms (homogeneous coordinates) for column vectors

$$F = \begin{bmatrix} M_{11} & M_{12} & M_{13} & T_x \\ M_{21} & M_{22} & M_{23} & T_y \\ M_{31} & M_{32} & M_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$F = \begin{bmatrix} M & T \\ 0 & 1 \end{bmatrix}$$

Inverse of 4D

$$F^{-1} = \begin{bmatrix} M^T & -M^T T \\ 0 & 1 \end{bmatrix}$$

Direction vectors vs. points

Direction vectors have w = 0

Points have w = 1

Therefore directions are unaffected by translation.

The projection into 3D of the 4D point P is:

$$\tilde{P} = \langle \frac{x}{w}, \frac{y}{w}, \frac{z}{w} \rangle$$

Thus, any scalar multiple of the 4D vector P represents the *same point* in 3D.

With this information:

Transforming tangents can be done as *direction* vectors, since they're often the difference between two vertices.

Transforming normals:

We have that $N \cdot T = 0$

We have that $T' = MT$

We require $N' \cdot T' = 0$

$$N' \cdot T' = (GN) \cdot (MT) = 0$$

$$(GN) \cdot (MT) = (GN)^T (MT) = N^T G^T MT = 0$$

Since $N^T T = N \cdot T = 0$, the above is true iff $G^T M = I$

Thus:

$$\boxed{G = (M^{-1})^T} \; for\ transforming\ normals$$

Vectors that must be transformed this way, like normals, are called **covariant**

Vectors that can simply use M are called **contravariant**.

Fortunately, for orthogonal M, G = M.

Quaternions:

$$q = \langle w, x, y, z \rangle = w + xi + yj + zk = s + \boldsymbol{v}$$

Multiplying the components looks like cross products:

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = -ji = k$$

$$jk = -kj = i$$

$$ki = -ik = j$$

$$q = \cos\left(\frac{\theta}{2}\right) + \boldsymbol{A}sin\left(\frac{\theta}{2}\right)$$

For unit axis A.

$$q_1 q_2 = s_1 s_2 - v_1 \cdot v_2 + s_1 v_2 + s_2 v_1 + v_1 \times v_2$$

$$\bar{q} = s - v$$

$$q\bar{q} = \bar{q}q = q \cdot q = \|q\|^2 = q^2$$

$$q^{-1} = \frac{\bar{q}}{q^2}$$

Rotation matrix corresponding to rotation of quaternion q:

$$R_q = \begin{bmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{bmatrix}$$

Lerp requires renormalization, and is slow near the **ends** and fast near the **middle**.

Slerp:

$$q(t) = \frac{\sin\big(\theta(1-t)\big)}{\sin(\theta)} q_1 + \frac{\sin(\theta t)}{\sin(\theta)} q_2$$

But $\theta = \arccos(q_1 \cdot q_2)$ so:

$$\sin(\theta) = \sqrt{1-\cos(\theta)^2} = \sqrt{1-(q_1 \cdot q_2)^2}$$

NOTE: q and –q are the same rotation, so select the sign of one of your q's such that $q_1 \cdot q_2 \geq 0$. This ensures interpolation takes the short path since the two quats are < 180 degrees (pi radians) apart.

Dual quats can be used to encode both translation and rotation. Each component is a dual number, a+eps*b, where eps^2=0.

Lines in 3D space:

Given two points P_1 and P_2 we can define the line that passes through these points as:

$$P(t) = (1-t)P_1 + tP_2$$

Where t ranges over ALL real numbers, with the segment in between the two points corresponding to t values between 0 and 1.

Rays start at a point and extend in a direction:

$$P(t) = S + tV$$

Where t is >=0. Note that the two formulations are equivalent if we let S = P1 and V = P2 – P1, and we remove the restriction on t.

Distance between a point and a line:

The distance from point Q to a line can be found by calculating the magnitude of the perpendicular component of Q – (any point on the line). So for a ray you could do Q – S. For a line you could do Q – P_1.

Then you need the perpendicular projection:

$$(Q - S) - proj_v(Q - S)$$

$$(Q - S) - \frac{(Q - S) \cdot V}{V^2} V$$

To get just the magnitude you can use Pythag with sides of d and $\|proj_v(Q - S)\|$ and hypot of $\|Q - S\|$.

$$d = \sqrt{\|Q - S\|^2 - \left\| \frac{(Q - S) \cdot V}{\|V\|^2} V \right\|^2}$$

$$d = \sqrt{\|Q - S\|^2 - \left[\frac{(Q - S) \cdot V}{\|V\|^2} \right]^2 \|V\|^2}$$

Distance from a point Q to a line defined by starting point S and direction V:

$$\boxed{d = \sqrt{\|Q - S\|^2 - \frac{[(Q - S) \cdot V]^2}{\|V\|^2}}}$$

Distance from a point Q to a line defined by points P1 and P2:

$$\boxed{d = \sqrt{\|Q - P_1\|^2 - \frac{[(Q - P_1) \cdot (P_2 - P_1)]^2}{\|P_2 - P_1\|^2}}}$$

Distance between two lines:

Two lines that are not parallel and do not intersect are called *skew* lines. We can find the minimum distance between points on skew lines.

We minimize the function $f(t_1, t_2) = \|P_1(t_1) - P_2(t_2)\|^2$ and set the partial derivatives w.r.t. t_1 and t_2 to 0, obtaining:

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \frac{1}{(V_1 \cdot V_2)^2 - V_1^2 V_2^2} \begin{bmatrix} -V_2^2 & V_1 \cdot V_2 \\ -V_1 \cdot V_2 & V_1^2 \end{bmatrix} \begin{bmatrix} (S_2 - S_1) \cdot V_1 \\ (S_2 - S_1) \cdot V_2 \end{bmatrix}$$

Plugging these t values back in will give us the minimum squared distance. Take the square root to get the distance itself. Note that if the direction vectors have unit length, things simplify.

If the denominator there is 0, the lines are parallel, and the distance can be found by considering point-to-line from S_1 to the second line, for example.

Kareem Omar
kareem_omar@naughtydog.com

Planes:

Given a 3D point P and a normal vector N, the plane passing through the point P and perpendicular to the direction N can be defined as the set of points Q such that $N \cdot (Q - P) = 0$. A vector from a point on the plane to another point on the plane, in other words, must be perpendicular to the normal vector. Well, if we distribute this, we get:

$$N \cdot Q + (-N \cdot P) = 0$$

$$N_x Q_x + N_y Q_y + N_z Q_z + (-N \cdot P) = 0$$

$$N_x x + N_y y + N_z z + (-N \cdot P) = 0$$

For any point Q = (x, y, z). This means we can write:

$$Ax + By + Cz + D = 0$$

Where

$$N = (A, B, C) \text{ and } \boxed{D = -N \cdot P}$$

Furthermore, we know that the shortest distance from the origin to the plane is along the N vector, so the projection of P onto N will give us that distance:

$$\|proj_N P\| = \|P\| cos(\alpha) = \frac{N \cdot P}{\|N\|} = \frac{|D|}{\|N\|}$$

To check whether a point is on the plane, or if not, on which side it's on, then, we can check where the point lies projected onto the normal vector. If it's further along than a point P on the plane, it's on the positive side of the plane. If it's not as far, it's on the negative side. And if it's the same, it's on the plane. So for arbitrary test point Q, we project Q onto N:

$$\|proj_N Q\| = \|Q\| cos(\alpha) = \frac{N \cdot Q}{\|N\|}$$

And then subtract our previously obtained result:

$$d = \frac{N \cdot Q}{\|N\|} - \frac{N \cdot P}{\|N\|}$$

$$d = \frac{N \cdot Q}{\|N\|} + \frac{D}{\|N\|}$$

$$\boxed{d = \frac{N \cdot Q + D}{\|N\|}}$$

Distance from a point to a plane.

And if we assume a unit normal vector, we get:

$$\boxed{d = N \cdot Q + D}$$

For any Q, where a d > 0 means on the positive side of the plane. d < 0 means negative side of the plane. d == 0 means on the plane (careful with floats!)

And therefore satisfying

$$N \cdot Q + D = 0$$

Is also the plane equation. This makes it very convenient to store the plane as:

$$L = \langle N, D \rangle$$

Considered as a homogeneous 4-vector, this means we have:

$$\boxed{d = L \cdot Q}$$

Where Q is given a w-coordinate of 1.

Recreating the closest point P to the origin, on the plane:

$$D = -N \cdot P$$

Taking P to be aN:

$$D = -N \cdot aN$$

$$D = -a(N \cdot N)$$

$$D = -a\|N\|^2$$

$$a = -\frac{D}{\|N\|^2}$$

$$P = aN = \boxed{-\frac{D}{\|N\|^2} N}$$

Makes sense: -D/||N|| (distance from the origin) in the direction of N.

Verify:

$$N \cdot P + D$$

$$N \cdot \left(-\frac{D}{\|N\|^2}N\right) + D$$

$$-D + D$$

$$0$$

Intersection of a line and a plane:

$$N \cdot P(t) + D = 0$$

$$N \cdot (S + tV) + D = 0$$

$$N \cdot S + t(N \cdot V) + D = 0$$

$$t = \frac{-(N \cdot S + D)}{N \cdot V}$$

$$P = S + \left(\frac{-(N \cdot S + D)}{N \cdot V}\right)V$$

If $N \cdot V = 0$, the line is parallel to the plane. It's in the plane itself if $N \cdot S + D = 0$, otherwise out of plane.

In the 4-vector representation we have:

$$t = -\frac{L \cdot S}{L \cdot V}$$

Where S has a 1 but V has a 0 as it's a direction, as desired.

Intersection of three planes:

$d = L \cdot Q = 0$ for all three planes

$$\begin{bmatrix} N_1 \\ N_2 \\ N_3 \end{bmatrix} Q = \begin{bmatrix} -D_1 \\ -D_2 \\ -D_3 \end{bmatrix}$$

$$Q = \begin{bmatrix} N_1 \\ N_2 \\ N_3 \end{bmatrix}^{-1} \begin{bmatrix} -D_1 \\ -D_2 \\ -D_3 \end{bmatrix}$$

Where $M = \begin{bmatrix} N_1 \\ N_2 \\ N_3 \end{bmatrix}$.

If M is singular (i.e. det M = 0), then the three planes do not intersect at a single point. This happens when the three normal vectors all lie in the same plane.

Intersection of two planes:

The line of intersection runs in direction $V = N_1 \times N_2$. To get a point on this line, create another plane with normal V that passes through the origin (i.e. $\langle V, 0 \rangle$). The intersection of all 3 planes is then guaranteed to exist and will generate a point on the line of intersection.

Transforming a plane:

Fortunately, they transform just fine as 4D vectors with inverse transpose:

$$L' = (F^{-1})^T L$$

View frustum:

X is to the right, y is up, and z is into the scene in DirectX (left-handed) or behind the camera in OpenGL (right-handed)

The projection plane is a distance e, the **focal length**, away from the camera, into the scene, such that it intersects the left and right frustum planes at $x = -1$ and $x = 1$. The **horizontal field of view** between the two extremes is α, such that the angle between the centerline and either is $\frac{\alpha}{2}$. Therefore we have:

$$e = \frac{1}{\tan\left(\frac{\alpha}{2}\right)}$$

Larger field of view, shorter focal length.

Vertical field of view:

$$\frac{e}{a} = \frac{1}{\tan\left(\frac{\beta}{2}\right)}$$

$$\beta = 2 \arctan\left(\frac{a}{e}\right)$$

Where a is the **aspect ratio** of the screen, height / width. (< 1!!) So as a gets larger, β gets larger.

Thus at the projection plane (distance e) from the camera, the four side planes of the view frustum carve out a rectangle with edges at $x = \pm 1$ and $y = \pm a$.

If, as with OpenGL, we must supply that rectangle out at n rather than at e, we can scale everything by $\frac{n}{e}$, resulting in the rectangle:

$$x = \pm\frac{n}{e}$$

$$y = \pm a\frac{n}{e}$$

View frustum planes / plane vectors in OpenGL:

Plane	$\langle N, D\rangle$
Near	$\langle 0,0,-1,-n\rangle$
Far	$\langle 0,0,1,f\rangle$
Left	$\langle\frac{e}{\sqrt{e^2+1}},0,-\frac{1}{\sqrt{e^2+1}},0\rangle$
Right	$\langle-\frac{e}{\sqrt{e^2+1}},0,-\frac{1}{\sqrt{e^2+1}},0\rangle$
Bottom	$\langle 0,\frac{e}{\sqrt{e^2+a^2}},-\frac{a}{\sqrt{e^2+a^2}},0\rangle$
Top	$\langle 0,-\frac{e}{\sqrt{e^2+a^2}},-\frac{a}{\sqrt{e^2+a^2}},0\rangle$

Perspective-Correct Interpolation:

40% of the way across a scanline along the projection plane isn't necessarily 40% of the way along a triangle, unfortunately, due to depth.

Note that z coordinates are linearly interpolated, not perspective interpolated, as explained below:

By similar triangles, a point out there at <x, z> hits the projection plane at:

$$\frac{p}{x} = -\frac{e}{z}$$

The reciprocal of the z coordinate is correctly interpolated in a linear manner across the face of a triangle:

$$\frac{1}{z_3} = \frac{1}{z_1}(1-t) + \frac{1}{z_2}t$$

Therefore, to interpolate a parameter b:

$$\frac{b_3}{z_3} = \frac{b_1}{z_1}(1-t) + \frac{b_2}{z_2}t$$

In other words, you can linearly interpolate b/z. So for a given point, graphics processors first calculate the linearly interpolated value $1/z_3$, then reciprocate it to get z_3, then multiply that by the linearly interpolated value of b/z.

Projection:

Ray cast from origin to point P:

$$x = -\frac{e}{P_z}P_x$$

$$y = -\frac{e}{P_z}P_y$$

However we need a different answer for z so it doesn't always end up at the projection plane (i.e. z = -e). This is so we can still have useful distance metrics for hidden surface removal. So instead, we use homogeneous coordinates and project vertices in 4D space.

This is called a **perspective projection** and it maps the view frustum into a cube representing **homogeneous clip space**.

In OpenGL, it's centered at the origin and goes from -1 to 1 in all 3 axes.

First, a 4x4 matrix places –z into w. Then a division by w produces a 3D point in **normalized device coordinates**.

The matrix:

$$M_{frustum} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note that this projection reflects z so homogeneous clip space is **left-handed in OpenGL**.

Applying this matrix to a point P gives:

$$P' = M_{frustum}P = \langle -x'P_z, -y'P_z, -z'P_z, -P_z \rangle$$

And it is the w coordinate, $-P_z$, which gets reciprocated and interpolated during rasterization.

Camera Space to normalized device coordinates (NDC):

$$+\infty \; to \; 0 \rightarrow \frac{f+n}{f-n} \; to \; \infty$$

$$0 \; to \; -n \rightarrow -\infty \; to \; -1$$

$$-n \; to \frac{-2fn}{f+n} \rightarrow -1 \; to \; 0$$

$$\frac{-2fn}{f+n} \; to \; -f \rightarrow 0 \; to \; 1$$

$$-f \; to \; -\infty \rightarrow 1 \; to \frac{f+n}{f-n}$$

The frustum could also have a far plane distance at infinity. This gives:

$$M_{infinite} = \lim_{f \to \infty} M_{frustum} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

And it is now possible to supply normal camera-space points (w = 1) or camera-space points with w = 0 to indicate they're at infinity, producing:

$$M_{infinite}Q = \begin{bmatrix} \frac{2n}{r-l}Q_x + \frac{r+l}{r-l}Q_z \\ \frac{2n}{t-b}Q_y + \frac{t+b}{t-b}Q_z \\ -Q_z \\ -Q_z \end{bmatrix}$$

Which results in a projected point with the maximum z coordinate (z=1) after division by w. This is useful for shadow rendering.

Orthographic Projection:

Depth values can be interpolated linearly, although we still flip z as we're still mapping [-n, -f] to [-1, 1].

$$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Extracting frustum planes:

$$L = [(M^{-1})^{-1}]^T L' = M^T L'$$

Clip-space plane vectors, and since this would get us two columns of M^T, it also gets us two rows of M:

Plane	$\langle N, D \rangle$	
Near	$\langle 0,0,1,1 \rangle$	$M_4 + M_3$
Far	$\langle 0,0,-1,1 \rangle$	$M_4 - M_3$
Left	$\langle 1,0,0,1 \rangle$	$M_4 + M_1$
Right	$\langle -1,0,0,1 \rangle$	$M_4 - M_1$
Bottom	$\langle 0,1,0,1 \rangle$	$M_4 + M_2$
Top	$\langle 0,-1,0,1 \rangle$	$M_4 - M_2$

Don't forget to normalize these.

Valid for any projection except the far plane of the infinite projection matrix. The previous table in terms of focal length and aspect ratio is more efficient.

Oblique near-plane clipping:

A perspective projection matrix can be modified so that the near plane is replaced by any camera-space clipping plane C, with $C_w < 0$, and modifying the far plane accordingly. This can allow free clipping at C for things like clipping at a reflection surface.

Newton's method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Converges quadratically.

Refinement of reciprocals:

$$x = \frac{1}{r}$$

$$\frac{1}{x} = r$$

$$\frac{1}{x} - r = 0$$

$$f(x) = \frac{1}{x} - r$$

$$f'(x) = -\frac{1}{x^2}$$

$$x_{n+1} = x_n - \frac{\frac{1}{x} - r}{-\frac{1}{x^2}}$$

$$x_{n+1} = x_n + \frac{\frac{1}{x} - r}{\frac{1}{x^2}}$$

$$x_{n+1} = x_n + x^2\left(\frac{1}{x} - r\right)$$

$$x_{n+1} = x_n + x - rx^2$$

$$x_{n+1} = 2x - rx^2$$

$$\boxed{x_{n+1} = x_n(2 - rx_n)} \text{ with } 0 < x_0 < \frac{2}{r}$$

Refinement of reciprocal square root:

$$x = \frac{1}{\sqrt{r}}$$

$$x^2 = \frac{1}{r}$$

$$\frac{1}{x^2} - r = 0$$

$$f(x) = \frac{1}{x^2} - r$$

$$f'(x) = -\frac{2}{x^3}$$

$$x_{n+1} = x_n - \frac{\frac{1}{x^2} - r}{-\frac{2}{x^3}}$$

$$x_{n+1} = x_n + \frac{\frac{1}{x^2} - r}{\frac{2}{x^3}}$$

$$x_{n+1} = x_n + x^3 \frac{\frac{1}{x^2} - r}{2}$$

$$x_{n+1} = x_n + \frac{\frac{x^3}{x^2} - x^3 r}{2}$$

$$x_{n+1} = x_n + \frac{x - x^3 r}{2}$$

$$x_{n+1} = x_n + \frac{1}{2}x - \frac{1}{2}x^3 r$$

$$x_{n+1} = \frac{3}{2}x_n - \frac{1}{2}x^3 r$$

$$x_{n+1} = \frac{1}{2}(3x_n - x^3 r)$$

$$\boxed{x_{n+1} = \frac{1}{2}x_n(3 - r x_n^2)} \text{ with } 0 < x_0 < \sqrt{\frac{3}{r}}$$

Intersection of a ray and a triangle:

For a triangle with vertices P_1, P_2, and P_3:

$$N = (P_1 - P_0) \times (P_2 - P_0)$$

Assuming a counter-clockwise winding order.

Form the plane this triangle resides in:

$$L = \langle N, -N \cdot P_0 \rangle$$

The ray (P(t) = S + tV) intersects this plane at:

$$t = -\frac{L \cdot S}{L \cdot V}$$

If $L \cdot V = 0$, no intersection occurs – the line is parallel to the plane, unless $L \cdot S = 0$ too, in which case the line is in the triangle.

Plug t back into the ray equation to get the point of intersection. Then we must check whether that point, P, lies inside the triangle. We do so by calculating the **barycentric coordinates** of P, which are a weighted average of the triangle's vertices:

$$P = w_0 P_0 + w_1 P_1 + w_2 P_2$$

Where $w_0 + w_1 + w_2 = 1$

And thus $w_0 = 1 - w_1 - w_2$

We get:

$$R = P - P_0$$

$$Q_1 = P_1 - P_0$$

$$Q_2 = P_2 - P_0$$

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \frac{1}{Q_1^2 Q_2^2 - (Q_1 \cdot Q_2)^2} \begin{bmatrix} Q_2^2 & -Q_1 \cdot Q_2 \\ -Q_1 \cdot Q_2 & Q_1^2 \end{bmatrix} \begin{bmatrix} R \cdot Q_1 \\ R \cdot Q_2 \end{bmatrix}$$

The point lies inside if all three weights are nonnegative, i.e. $w_1 \geq 0, w_2 \geq 0, w_1 + w_2 \leq 1$.

These weights can also be used for interpolation of vertex properties.

Intersection of a ray and a box, in the box's object space:

$$x = 0 \qquad x = r_x$$

$$y = 0 \qquad y = r_y$$

$$z = 0 \qquad z = r_z$$

We only need to check at most 3 planes. We should determine which planes to test by examining the components of V one at a time. For x: if $V_x = 0$, the ray cannot intersect either of the 'x' planes because they are at constant x, and so is the ray. If $V_x > 0$ we don't need to test $x = r_x$ as it's the back plane. If $V_x < 0$, we don't need to test $x = 0$ as it's the front plane.

Examine all three components in this way to determine the up to 3 planes that must be tested. Then test them:

$$t = -\frac{L \cdot S}{L \cdot V}$$

Plane	L	t-test
$x = 0$	$\langle 1,0,0,0 \rangle$	$t = -\frac{S_x}{V_x}$
$x = r_x$	$\langle 1,0,0,-r_x \rangle$	$t = \frac{r_x - S_x}{V_x}$
$y = 0$	$\langle 0,1,0,0 \rangle$	$t = -\frac{S_y}{V_y}$

$y = r_y$	$\langle 0,1,0,-r_y \rangle$	$t = \frac{r_y - S_y}{V_y}$
$z = 0$	$\langle 0,0,1,0 \rangle$	$t = -\frac{S_z}{V_z}$
$z = r_z$	$\langle 0,0,1,-r_z \rangle$	$t = \frac{r_z - S_z}{V_z}$

And to lie within the face of the box, the point must be between 0 and r in the *other two dimensions*.

If you find an intersection you're done as no closer intersection could take place.

Intersection of a ray and a sphere in object space:

A sphere of radius r centered at the origin is described by the equation:

$$x^2 + y^2 + z^2 = r^2$$

We substitute in the ray for x, y, and z, and look for a t, or range of t, that makes it true. We get:

$$a = V^2$$

$$b = 2(S \cdot V)$$

$$c = S^2 - r^2$$

The discriminant, $D = b^2 - 4ac$, tells us what we need to know:

If $D < 0$, no intersection occurs.

If $D = 0$, the ray is tangent.

If $D > 0$, the ray enters the sphere, spends some time inside, and then exits at another point of intersection.

If there are two, we generally want the smaller t, so:

$$t = \frac{-b - \sqrt{D}}{2a}$$

Intersection of a ray and an ellipsoid in object space:

$$x^2 + m^2y^2 + n^2z^2 = r^2$$

m is the ratio of the x semiaxis to the y semiaxis, and n is the ratio of the x semiaxis to the z semiaxis.

We get:

$$a = V_x^2 + m^2 V_y^2 + n^2 V_z^2$$

$$b = 2\left(S_x V_x + m^2 S_y V_y + n^2 S_z V_z\right)$$

$$c = S_x^2 + m^2 S_y^2 + n^2 S_z^2 - r^2$$

Again, the discriminant indicates the manner of intersection and the above formula for t can find the (closer) intersection.

Intersection of a ray and a cylinder:

The lateral surface of an elliptical cylinder whose radius on the x axis is r, whose radius on the y axis is s, whose height is h, and whose base is centered on the origin of the x-y plane is described by the equation:

$$x^2 + m^2 y^2 = r^2$$

$$0 \le z \le h$$

Where $m = \frac{r}{s}$.

We get:

$$a = V_x^2 + m^2 V_y^2$$

$$b = 2\left(S_x V_x + m^2 S_y V_y\right)$$

$$c = S_x^2 + m^2 S_y^2 - r^2$$

Check the discriminant. If there are two solutions, you must test both to see whether they satisfy $0 \le z \le h$.

Normal vectors from surface implicit functions:

Suppose that $f(x, y, z)$ represents a surface S, so that $f = 0$ for any point on S. For example, an ellipsoid might look like:

$$f(x, y, z) = \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1$$

If we have a curve C on S:

$$C = \langle x(t), y(t), z(t) \rangle$$

Then the tangent vector to C, and thus also S, is:

$$T = \langle \frac{d}{dt} x(t), \frac{d}{dt} y(t), \frac{d}{dt} z(t) \rangle$$

But since f = 0 all over the surface, for any t, $\frac{df}{dt}$ must be 0 everywhere on the curve, so:

$$0 = \frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} + \frac{\partial f}{\partial z}\frac{dz}{dt} = \langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \rangle \cdot T$$

So the dot product of that vector operator with T is always 0. Therefore it must always be normal to T, and thus normal to the surface S. This is the **gradient** of **f** at the point <x, y, z> and is usually written:

$$\nabla f(x, y, z)$$

Where the symbol ∇ is the **del** operator defined by:

$$\nabla = i\frac{\partial}{\partial x} + j\frac{\partial}{\partial y} + k\frac{\partial}{\partial z}$$

So then the normal to a surface defined by f is:

$$\boxed{N = \nabla f(x, y, z) = \langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \rangle}$$

Reflection and refraction vectors:

Reflection vector (specular reflection):

To go from L (toward light) to R (reflection out from surface), both normalized, we want to subtract out double the component perpendicular to the surface normal. The first subtraction gets you N, the second flips it across to R:

$$R = L - 2perp_N L$$

$$R = L - 2[L - (N \cdot L)N]$$

$$R = L - 2L + 2(N \cdot L)N$$

$$\boxed{R = 2(N \cdot L)N - L}$$

Refraction vector (Snell's law):

$$\eta_L \sin(\theta_L) = \eta_T \sin(\theta_T)$$

Air is 1.0003 index of refraction

Water is 1.33 index of refraction

We have L (toward light) and N (surface normal), both normalized. We want T, the transmitted vector out.

The component of T parallel to the normal is just:

$$-Ncos(\theta_T)$$

The component of T perpendicular to the normal is:

$$-Gsin(\theta_T)$$

Where G is the unit vector parallel to $perp_N L$. Since L has unit length, $\|perp_N L\| = \sin(\theta_L)$, so:

$$G = \frac{perp_N L}{\sin(\theta_L)} = \frac{L - (N \cdot L)N}{\sin(\theta_L)}$$

And T is now the sum of these parts:

$$T = -Ncos(\theta_T) - Gsin(\theta_T)$$

$$T = -Ncos(\theta_T) - \frac{sin(\theta_T)}{\sin(\theta_L)}[L - (N \cdot L)N]$$

$$T = -Ncos(\theta_T) - \frac{\eta_L}{\eta_T}[L - (N \cdot L)N]$$

$$T = -N\left(\sqrt{1 - \sin^2(\theta_T)}\right) - \frac{\eta_L}{\eta_T}[L - (N \cdot L)N]$$

$$T = -N\left(\sqrt{1 - \frac{\eta_L^2}{\eta_T^2}\sin(\theta_L)^2}\right) - \frac{\eta_L}{\eta_T}[L - (N \cdot L)N]$$

$$T = -N\left(\sqrt{1 - \frac{\eta_L^2}{\eta_T^2}(1 - \cos(\theta_L)^2)}\right) - \frac{\eta_L}{\eta_T}[L - (N \cdot L)N]$$

$$T = -N\left(\sqrt{1 - \frac{\eta_L^2}{\eta_T^2}(1 - (\mathrm{N} \cdot \mathrm{L})^2)}\right) - \frac{\eta_L}{\eta_T}[L - (N \cdot L)N]$$

$$T = -N\left(\sqrt{1 - \frac{\eta_L^2}{\eta_T^2}(1 - (\mathrm{N} \cdot \mathrm{L})^2)}\right) - \frac{\eta_L}{\eta_T}L + \frac{\eta_L}{\eta_T}(N \cdot L)N$$

$$\boxed{T = \left(\frac{\eta_L}{\eta_T}(N \cdot L) - \sqrt{1 - \frac{\eta_L^2}{\eta_T^2}(1 - (\mathrm{N} \cdot \mathrm{L})^2)}\right)N - \frac{\eta_L}{\eta_T}L}$$

If the quantity inside the radical is negative, $\sin(\theta_L) > \frac{\eta_T}{\eta_L}$ (therefore possible only if $\eta_L > \eta_T$). In this case we have **total internal reflection** and we should use specular reflection. That's the **critical angle**, and any angle greater than that will cause total internal reflection. At the critical angle, the ray will theoretically run along the surface tangent.

Point light:

$$C = \frac{1}{k_c + k_l d + k_q d^2} C_0$$

Where C_0 is the color of the light and k are the constant, linear, and quadratic attenuation constants. d is $\|P - Q\|$, the distance to the light source.

Can also have a range to limit its effect.

Spot light:

$$C = \frac{\max(-R \cdot L, 0)^p}{k_c + k_l d + k_q d^2} C_0$$

R is the spotlight direction. L is the normalized direction pointing from Q toward the light source:

$$L = \frac{P - Q}{\|P - Q\|}$$

The exponent p controls how concentrated the spot light is. Larger p means a tighter beam.

Diffuse reflection:

A diffuse surface is one for which part of the light incident on a point on the surface is scattered in random directions. On average, this means that a certain color of lift, the surface's diffuse reflection

color, is reflected uniformly in every direction. This is called **Lambertian** reflection, and because it's assumed equal in all directions, it's not dependent on the position of the observer.

Note that as a light shaft with area A impinges on a surface, if the surface is not perpendicular to the light direction, the area illuminated is $\frac{A}{\cos(\theta)}$, and thus the light intensity decreases (it's multiplied by $\cos(\theta)$).

This $\cos(\theta)$ value is also given by the dot product between the normal vector N and the unit direction to the light source, L. A negative dot product means the surface is facing away and should not be illuminated at all, so we should clamp the dot product to go no lower than 0.

$$K_{diffuse} = D\left(A + \sum_{i=1}^{n} C_i \max(N \cdot L_i, 0)\right)$$

Specular reflection:

Highlight due to reflection in the mirror-like manner. Depends on viewer position whether they see this or not.

$$K_{specular} = S\sum_{i=1}^{n} C_i \, max(R_i \cdot V, 0)^m (N \cdot L_i > 0)$$

Large specular power (m) means a sharp highlight that quickly fades out.

However, R is somewhat expensive to calculate as we must use the specular reflection equation:

$$R = 2(N \cdot L)N - L$$

So we also have the halfway vector approximation. The halfway vector is halfway between the viewer and the light source, and so the reflections will be strongest when it's near the normal vector. Thus we can write:

$$H_i = \frac{L_i + V}{\|L_i + V\|}$$

$$K_{specular,halfway} = S\sum_{i=1}^{n} C_i \max(N \cdot H_i, 0)\,(N \cdot L_i > 0)$$

Texture mapping:

At each point on a surface, a texel is looked up and combined with the lighting. In the simplest case, you can use it to module the diffuse reflection color:

$$K_{diffuse} = DT\left(A + \sum_{i=1}^{n} C_i \max(N \cdot L_i, 0)\right)$$

Similarly a **gloss map** could modulate the specular:

$$K_{specular} = SG \sum_{i=1}^{n} C_i \, max(R_i \cdot V, 0)^m (N \cdot L_i > 0)$$

Texture coordinates:

Texture coordinates are either precomputed and stored with each vertex or calculated at runtime. They are then interpolated using 5.37:

$$\frac{b_3}{z_3} = \frac{b_1}{z_1}(1-t) + \frac{b_2}{z_2}t$$

Texture maps are typically 1-4 dimensions and use the variables **s, t, p, q**. They tend to go from 0 to 1.

Typically t is “up”, s is to the “right”, and p is depth into the page, for a cube.

u is used as the interpolation parameter instead of t or β.

We can perform projective maps by interpolating both s (or t or p) and q:

$$s = \frac{s_3}{q_3} = \frac{(1-u)\frac{s_1}{z_1} + u\frac{s_2}{z_2}}{(1-u)\frac{q_1}{z_1} + u\frac{q_2}{z_2}}$$

Z buffer:

The z- and w-buffers store coordinates that are expressed in clip space. But in terms of view space coordinates, the z-buffer stores 1/z while the w-buffer stores z.

Rendering pipeline:

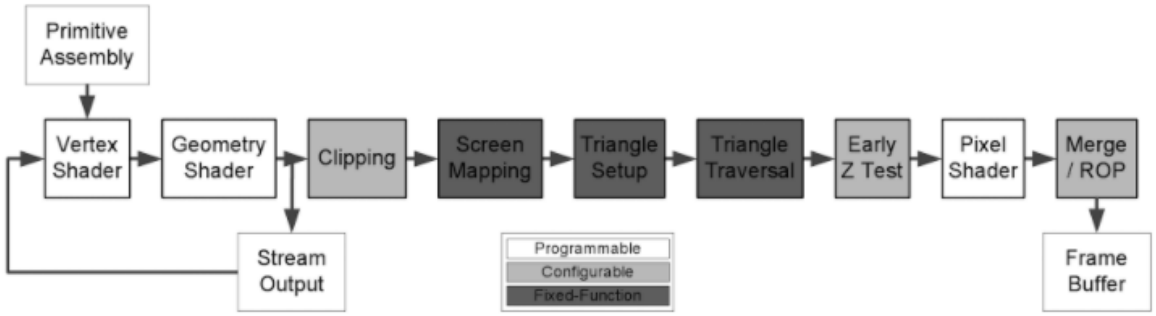


Figure 10.44. The geometry processing and rasterization stages of the rendering pipeline, as implemented by a typical GPU. The white stages are programmable, the light grey stages are configurable, and the dark grey boxes are fixed-function.

PS4 has fast shader resource tables (SRTs) allowing the GPU to access them via the high-speed garlic bus.

Cube maps:

Useful for environment maps for reflections. Store the camera direction reflection and sample using the component of greatest absolute value to determine which face to sample.

Normalization cube maps:

Stores vectors encoded as colors:

$$R = \frac{x+1}{2}$$

$$G = \frac{y+1}{2}$$

$$B = \frac{z+1}{2}$$

If the cube map retrieval is done in hardware, this means you can renormalize a vector just by querying the cubemap with your (not current normalized) vector.

Filtering and mipmaps:

Bilinear interpolation:

$$i = \left\lfloor ws - \frac{1}{2} \right\rfloor$$

$$j = \left\lfloor ht - \frac{1}{2} \right\rfloor$$

$$\alpha = frac\left(ws - \frac{1}{2}\right)$$

$$\beta = frac\left(ht - \frac{1}{2}\right)$$

$$\mathcal{T} = (1-\alpha)(1-\beta)\mathcal{T}_{\langle i,j\rangle} + \alpha(1-\beta)\mathcal{T}_{\langle i+1,j\rangle} + (1-\alpha)\beta\mathcal{T}_{\langle i,j+1\rangle} + \alpha\beta\mathcal{T}_{\langle i+1,j+1\rangle}$$

Mipmap with each at half the size in each dimension, i.e. ¼ the pixels, so we have:

$$1 + \frac{1}{4} + \frac{1}{16} + \cdots = \frac{4}{3}$$

A texture map only 1/3 larger than the original texture map.

Doing the math:

Let $u(x,y) = 2^n s(x,y)$ and $v(x,y) = 2^m t(x,y)$. The level-of-detail parameter λ is determined by calculating:

$$\rho_x = \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}$$

$$\rho_y = \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}$$

$$\lambda = \log_2\left(\max(\rho_x, \rho_y)\right)$$

When using bilinear filtering, λ is rounded to the nearest integer and clamped to the range [0, max(n, m)]. Four texture samples are then fetched from the corresponding mipmap level and blended via blerp.

Trilinear filtering:

Prevent jarring changes in mipmap level by sampling from floor(λ) and floor(λ) +1 and then:

$$\mathcal{T} = \big(1 - frac(\lambda)\big)\mathcal{T}_1 + frac(\lambda)\mathcal{T}_2$$

Mipmapping for 1D and 3D maps consider 1 or 3 texture coordinates in the lambda calculation. For cubemaps it operates independently for each of the six 2D faces.

Emission:

Some objects may emit light, potentially modulated by an emission map that determines the color and intensity of the glow at each point on a surface:

$$K_{emission} = EM$$

Unlike just sticking this map into the ordinary texture, the emission map is unaffected by the direction of the surface normal (or anything else).

Shading models:

To apply lighting, we need the surface normal at each vertex (or perhaps each pixel). For a single triangle we can use the cross product as we've done before, assuming a counter-clockwise winding order.

Then for a vertex you could average all the normal vectors of all the triangles that share that vertex (just add and renormalize).

If you want the average to be weighted by the area of the triangles, don't normalize the normal vectors first. Might look better.

Gouraud shading:

This is the vertex interpolation technique. So at each vertex we calculate:

$$K_{primary} = E + DA + D\sum_{i=1}^{n} C_i \max(N \cdot L_i, 0)$$

This includes emission (potentially also an emission map) and diffuse.

$$K_{secondary} = S\sum_{i=1}^{n} C_i \max(N \cdot H_i, 0)^m \ (N \cdot L_i > 0)$$

This includes specular (possible also a gloss map).

Final pixel color is then:

$$K = K_{primary} \circ \mathcal{T}_1 \circ \mathcal{T}_2 \circ \ldots \circ \mathcal{T}_k + K_{secondary}$$

Blinn-Phong shading:

Interpolate vertex normal (renormalize!!) N, direction to light source L, direction to viewer V, and intensity of a light source C_i, then evaluate at **each pixel**. Calculate the halfway vector at each pixel, too:

$$H_i = \frac{L_i + V}{\|L_i + V\|}$$

Then we have, for each pixel:

$$K = K_{emission} + K_{diffuse} + K_{specular}$$

$$= EM + DTA + \sum_{i=1}^{n} C_i[DT \max(N \cdot L_i, 0) + SG \max(N \cdot H_i, 0)^m (N \cdot L_i > 0)]$$

Way better at specular. Gouraud looks kinda funky if specular is supposed to land in the middle of a triangle. Renormalize N or specular will darken in the middle of tris!

Bump mapping:

Because we've been interpolating vertex normals, even in Phong shading, we're limited in how we vary illumination. Bump mapping lets us encode additional vectors (relative to the existing interpolated normal vector) to add per-pixel, so for example <0, 0, 1> represents an unmodified normal.

A bump map is typically constructed by extracting normal vectors from a **height map**, whose contents represent the height at each pixel. We first calculate tangents in s and t using differences in height between adjacent pixels:

$$S(i,j) = \langle 1,0, aH(i+1,j) - aH(i-1,j) \rangle$$

$$T(i,j) = \langle 0,1, aH(i,j+1) - aH(i,j-1) \rangle$$

Where a is a scale factor that can be used to vary how pronounced the perturbations will be.

Then:

$$N(i,j) = \frac{S(i,j) \times T(i,j)}{\|S(i,j) \times T(i,j)\|} = \frac{\langle -S_z, -T_z, 1 \rangle}{\sqrt{S_z^2 + T_z^2 + 1}}$$

In order to make <0,0,1> correspond to no change, we need to apply the bump map in a coordinate space at each vertex, in which the vertex normal is *already* at +z. The tangent vectors need to be sensible too – aligned with the texture space of the bump map.

The result is **tangent space** or **vertex space**. Once this is established, we no longer need the normal vectors at all! We transform L into tangent space and then dot it with the bump map sample.

As for that alignment – we need the x axis of tangent space to be aligned with the s direction in the bump map, and the y axis to be aligned to the t direction in the bump map.

That is, if Q is a point inside a triangle, we want to be able to write:

$$Q - P_0 = (s - s_0)T + (t - t_0)B$$

Where T and B are tangent vectors aligned to the texture map, and the 0s are one of the vertices.

B stands for **bitangent** and it represents the direction orthogonal to the tangent vector T in the tangent plane to the surface.

So for a triangle we let:

$$Q_1 = P_1 - P_0$$

$$Q_2 = P_2 - P_0$$

$$\langle s_1, t_1 \rangle = \langle s_1 - s_0, t_1 - t_0 \rangle$$

$$\langle s_2, t_2 \rangle = \langle s_2 - s_0, t_2 - t_0 \rangle$$

We need to solve for T and B:

$$Q_1 = s_1 T + t_1 B$$

$$Q_2 = s_2 T + t_2 B$$

Solving this gives us the tangent vectors for a triangle. We average like with normal vectors to get them for a vertex.

To invert the matrix it might be good to Gram-Schmidt T, B, and N first. This will let us transform lights from object to tangent space.

No need to store B since $N \times T'$ can produce mB', where $m = \pm 1$ represents the handedness of tangent space. This DOES need to be stored, possibly as the w coordinate of T'.

Physical reflection model:

Bidirectional reflectance distribution functions (BRDF) take L and R and return the amount of incident light from L that goes in the direction R.

First some **radiometry**:

Radiant power (energy per unit time) emitted by a light source or received by a surface is called **flux** and is measured in Watts.

Radiant power per unit area is called **flux density** and is measured in Watts per square meter.

The flux density emitted by a surface is called the surface's **radiosity**, and the flux density incident on a surface is called the **irradiance** of the light.

The power emitted and received can be the same, but the flux density will still differ due to a wider incident area:

$$\Phi_I = \Phi_E (N \cdot L)$$

Solid angle:

$$\theta = \frac{l}{r} \text{ radians, max of } \frac{2\pi r}{r} = 2\pi$$

$$\omega = \frac{A}{r^2} \text{ steradians, max of } \frac{4\pi r^2}{r^2} = 4\pi$$

Radiance is the flux density per unit solid angle, and is measured in Watts per square meter per steradian.

Cook-Torrance Illumination:

Produces realistic specular by treating a surface as composed of planar **microfacets**.

Fresnel factors:

Incident light can be decomposed into components polarized w.r.t. the plane containing N and L. The component parallel to this plane is p-polarized, and the component perpendicular is s-polarized. Fresnel factors give the fractions of the amount of light reflected for these components.

Anisotropic microfacet distributions allow materials such as brushed metal, hair, and fabrics to have different degrees of roughness in different directions.

Geometrical attenuation factor:

We can estimate the amount of light incident on a microfacet that's blocked by adjacent ones first (or after reflection), which results in a slight darkening of specular and increase in diffuse.

Visibility Determination:

Principal Component Analysis:

A technique that allows us to find a coordinate space in which a set of data composed of multiple variables, such as the x, y, and z coordinates stored in an array of vertex positions, can be separated into uncorrelated components. The primary principal component of the data is represented by the direction in which the data varies the most.

We first calculate the mean position m:

$$m = \frac{1}{N}\sum_{i=1}^{N} P_i$$

We then construct a 3x3 matrix C, our **covariance matrix**:

$$C = \frac{1}{N}\sum_{i=1}^{N} (P_i - m)(P_i - m)^T$$

This makes C a symmetric matrix with entries representing the correlation between pairs of coordinates. 0 indicates no correlation. We want only the diagonal terms to be nonzero, so we can diagonalize C. Find the eigenvalues, then eigenvectors. The unit-length eigenvectors constitute R, S, and T, the natural axes. R is the **principal axis**. It corresponds to the largest eigenvalue.

To find the extents of our points P in RST space, we dot each P with R, S, and T to get their projections. The bounding box planes are then:

$$\langle R, -\min(P_i \cdot R)\rangle \qquad \langle -R, \max(P_i \cdot R)\rangle$$

$$\langle S, -\min(P_i \cdot S)\rangle \qquad \langle -S, \max(P_i \cdot S)\rangle$$

$$\langle T, -\min(P_i \cdot T)\rangle \qquad \langle -T, \max(P_i \cdot T)\rangle$$

We assign to the scalars a, b, and c the average extent in the R, S, and T directions:

$$a = \frac{\min(P_i \cdot R) + \max(P_i \cdot R)}{2}$$

$$b = \frac{\min(P_i \cdot S) + \max(P_i \cdot S)}{2}$$

$$c = \frac{\min(P_i \cdot T) + \max(P_i \cdot T)}{2}$$

The planes that divide the box are:

$$\langle R, -a\rangle$$

$$\langle S, -b\rangle$$

$$\langle T, -c\rangle$$

And the center is:

$$Q = aR + bS + cT$$

Bounding spheres:

Useful because the test is faster. Approximate by calculating principal axis R and finding min and max dotted with R. Construct a sphere with center Q and radius r:

$$Q = \frac{P_k + P_l}{2}$$

$$r = \|P_k - Q\|$$

Unfortunately, this may not enclose all points. Test them all. A point lies outside if:

$$\|P_i - Q\|^2 > r^2$$

The point on the edge that can remain the same is G:

$$G = Q - r\,\frac{P_i - Q}{\|P_i - Q\|}$$

$$Q' = \frac{G + P_i}{2}$$

$$r' = \|P_i - Q'\|$$

Bounding ellipsoids:

Get bounding box. Scale to make it a cube. Get bounding sphere. Scale back out into ellipsoid.

Bounding cylinder:

Represented by radius and two points corresponding to the centers of its endcaps. Cylinder should run along the principal axis, R. Now to find the radius of the cylinder:

$$H_i = P_i - (P_i \cdot R)R$$

Dot all H with S and find the min and max. Then:

$$Q = \frac{H_k + H_l}{2}$$

$$r = \|H_k - Q\|$$

Then check all the points. If it happens that:

$$\|H_i - Q\|^2 > r^2$$

Then expand the bounding circle:

$$G = Q - r\,\frac{H_i - Q}{\|H_i - Q\|}$$

$$Q' = \frac{G + H_i}{2}$$

$$r' = \|H_i - Q'\|$$

End points:

$$Q_1 = Q + \min(P_i \cdot R)\, R$$

$$Q_2 = Q + \max(P_i \cdot R)\, R$$

Bounding volume tests:

These are generally accomplished by moving out the planes of the view frustum by an appropriate amount, then doing a simple point or line intersection test.

Bounding sphere test:

Dot Q, in 4D, with the frustum planes. If any dot product is less than or equal to –r, the sphere does not intersect the view frustum and the object bounded by it can be culled from the visible set.

Other tests:

Point tests against effective radius $-r_{eff}$, or line tests (cull if BOTH endpoints are <= $-r_{eff}$).

Quadtrees/octrees:

Recursively subdivide world into quadrants or octants, with each containing a pointer back up to its parent, and each world object contains a pointer to the smallest octant that fully contains it.

To test, we start at the root and check visibility. We ignore the children of any node that's not visible. Also, once we calculate our five effective radii (one for the near and far planes, 4 for the side planes), the effective radius for a level deeper is just $r_{eff}/2$.

BSP tree:

Arbitrarily oriented planes. Traditionally an existing poly makes up the **splitting plane**. Modern approach is to create a splitting for each object, aligned perpendicular to the object's principal axis T (the smallest dimension of the bounding box). Objects in both half-spaces are added to both instead of being split up.

Test planes by transforming to homogeneous clip space and testing the vertices of the view frustum with matching component signs (max) or opposite signs (min). If d_max <= 0, frustum is entirely negative. Cull the positive side. If d_min >= 0, frustum is entirely positive. Cull the negative side. If neither, we can't cull anything.

Portal systems:

Adding planes:

Acute angles of meeting between reduced view frustum planes can cause bounding volume tests to succeed spuriously too often. Detect and add another clipping plane:

$$A = N_1 + N_2$$

$$B = N_1 \times N_2$$

$$N_3 = \frac{A - (A \cdot B)B}{\|A - (A \cdot B)B\|}$$

Projection matrix modification offset:

It is possible to modify the matrix so that the only effect is to offset projected depth values by a constant $\frac{\epsilon(f+n)}{f-n}$, without affecting the projected x or y. We change the second-from-the-bottom-and-left term:

$$-\frac{f+n}{f-n}$$

Into:

$$-\frac{(1+\epsilon)(f+n)}{f-n}$$

Decal construction:

We have our center point P on a surface and the normal N at that point. T determines orientation. This means we have an oriented tangent plane for our decal. Carve a rectangle out of it using four boundary planes parallel to N. Letting w and h be the width and height of the decal:

$$B = N \times T$$

$$left = \left(T, \frac{w}{2} - T \cdot P\right)$$

$$right = \left(-T, \frac{w}{2} + T \cdot P\right)$$

$$bottom = \left(B, \frac{h}{2} - B \cdot P\right)$$

$$top = \left(-B, \frac{h}{2} + B \cdot P\right)$$

Also probably want to clip to front and back planes to avoid geometry above or below getting affected:

$$front = (-N, d + N \cdot P)$$

$$back = (N, d - N \cdot P)$$

Where d is the maximum distance that any vertex in the decal may be from the tangent plane passing through P.

For nearby surfaces, if $N \cdot M$, where M is the unit normal of the tri, is positive, we keep it, clip it to the planes, and store it in a new triangle mesh.

Polygon clipping:

When a polygon is clipped against a plane, new vertices are added where edges intersect the plane, and vertices lying on the negative side of the plane are removed.

Each clipping operation can add 1 vertex, so clipping a triangle against 6 planes can leave it with up to 9 vertices. For portals we have verts in the plane. For general polygons we just categorize the verts as either positive or negative side of the plane. If all negative, discard the poly. Otherwise, visit every pair of neighboring verts and see if their edge intersects the clipping plane. If so, add a vert at that location.

Billboarding:

Unconstrained quads:

Let R and U be the world-space right and up directions of the camera, C be the current camera position, and P be the center of the billboard.

Point toward camera plane (cheap):

$$X = \left(\frac{w}{2}\cos(\theta)\right)R + \left(\frac{w}{2}\sin(\theta)\right)U$$

$$Y = \left(-\frac{h}{2}\sin(\theta)\right)R + \left(\frac{h}{2}\cos(\theta)\right)U$$

Point toward camera (expensive):

$$Z = \frac{C - P}{\|C - P\|}$$

$$A = \frac{U \times Z}{\|U \times Z\|}$$

$$B = Z \times A$$

$$X = \left(\frac{w}{2}\cos(\theta)\right)A + \left(\frac{w}{2}\sin(\theta)\right)B$$

$$Y = \left(-\frac{h}{2}\sin(\theta)\right)A + \left(\frac{h}{2}\cos(\theta)\right)B$$

Verts:

$$Q_1 = P + X + Y$$

$$Q_2 = P - X + Y$$

$$Q_3 = P - X - Y$$

$$Q_4 = P + X - Y$$

Constrained quads:

$$X = \langle P_y - C_y, C_x - P_x, 0 \rangle$$

$$Q_1 = P + \frac{w}{2}\frac{X}{\|X\|} + \langle 0,0,\frac{h}{2} \rangle$$

$$Q_2 = P - \frac{w}{2}\frac{X}{\|X\|} + \langle 0,0,\frac{h}{2} \rangle$$

$$Q_3 = P - \frac{w}{2}\frac{X}{\|X\|} - \langle 0,0,\frac{h}{2} \rangle$$

$$Q_4 = P + \frac{w}{2}\frac{X}{\|X\|} - \langle 0,0,\frac{h}{2} \rangle$$

Polyboards:

Used to give a polyline some thickness r.

Polygon reduction

T-Junction elimination

Triangulation:

Polys with n vertices become n – 2 triangles.

For convex polys, pick a vertex and connect edges to every nonadjacent vertex to create a **triangle fan**.

However, even for nominally convex polys, after welding and T-junction elimination, there may be collinear or nearly collinear points, so we must treat it as concave.

To do this we search for a set of 3 consecutive verts for which the formed triangle is not degenerate, is not wound the wrong way, and does not contain any other vertices. This is called an **ear**. It becomes a triangle and its middle vertex is removed from consideration. Stop when 3 verts remain.

Shadows:

Shadow mapping / shadow buffering, in which you generate a depth image from the location of a light source

or

stencil shadows / shadow volumes, in which you derive volumes of space for which light is blocked

Shadow-casting set: superset of view frustum / visible set. If light source is in view frustum, we're done. If outside, we extend the view frustum to the convex hull from the view frustum to the light source.

Shadow mapping:

Render from the perspective of a light source to generate a **shadow map** of just depth information representing the depth of each pixel in the scene with respect to the light source. Good for things that use the alpha test to cut holes, like leaves on a tree.

Orthographic projection for directional lights (aka infinite lights)!

During main rendering, vertices are transformed into the light space, have the projection applied, and then are used to s,t-lookup in the shadow map. If the depth of the vertex in the shadow space is greater, the point lies in shadow, since it cannot be seen from the light source.

Self-shadowing / shadow acne:

Due to limited precision of the depth buffer, some points can appear higher in the shadow map than when being main rendered, so they end up being counted as in shadow.

Stencil shadows:

Cheaper method: render light-independent stuff first, like ambient illumination, emission, and environment mapping. Then render the front faces of the shadow volume, incrementing on depth-test pass. The render the back faces, decrementing on depth-test pass. Nonzero stencil values are in shadow.

2 problems: need infinite extrude distance. Can use infinite far plane projection to solve.

And, can't have camera in shadow volume or near plane clipping the volume. To solve, cap the shadow volume, then render back, incrementing on depth-test fail. Then render front, decrementing on depth-test fail.

Silhouette determination:

First create an edge list. This is done by finding every edge where vertex indices are ascending. We find the second triangle attached to each edge by looking for the same pair of vertices connected in the other direction. Then:

$$N = (V_{i2} - V_{i1}) \times (V_{i3} - V_{i1})$$

Plane of the triangle:

$$F = \langle N_x, N_y, N_z, -N \cdot V_{i1} \rangle$$

Let L be the light position. For directional light, $L_w = 0$. Otherwise, $L_w \neq 0$. A triangle faces the light source if $F \cdot L > 0$.

The silhouette is equal to the set of edges shared by one triangle facing the light and one facing away.

Scissor optimization:

Light sources often have a max range, r. We can construct 4 planes going through the camera location and tangent to the sphere of influence of the light, then intersect these planes with the image plane and only render inside these intersections.

Curves and surfaces:

Cubic curve:

$$\boldsymbol{Q}(t) = \boldsymbol{a} + \boldsymbol{b}t + \boldsymbol{c}t^2 + \boldsymbol{d}t^3$$

$$Q(t) = CT(t)$$

Where

$$T(t) = \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

$$Q'(t) = C\frac{d}{dt}T(t) = C\begin{bmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \end{bmatrix}$$

A long curving path can be composed of smaller cubic pieces connected at their endpoints. At the points where two adjacent pieces join, there are notions of **parametric continuity** and **geometric continuity**. The symbol C^n is used to represent n-th order parametric continuity, and the symbol G^n is used to represent n-th order geometric continuity. The two curves are said to have C^1 continuity if their tangent vectors are equal at the join point. If they point in the same direction but have different magnitudes, the curves have G^1 continuity. In general, two curves meet with C^n continuity if their nth derivatives are equal, and G^n continuity if their nth derivatives are non-zero and point in the same direction but with different magnitudes. C^n continuity implies G^n continuity unless the nth derivatives are zero. C^0 and G^0 are equivalent and just mean the curves share a common endpoint.

We can express a cubic curve as a weighted sum of four geometrical constraints, weighted by 4 **blending functions** that can be written in matrix form. Thus we have:

$$Q(t) = GMT(t)$$

Where G is called the **geometry matrix** and M is called the **basis matrix**. Typically a class of curves is defined by a (constant) M, and G defines a particular curve within the class.

Hermite curves:

Defined by two endpoints and the tangents at those endpoints.

$$G_H = [P_1 \quad P_2 \quad T_1 \quad T_2]$$

$$M_H = \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Continuity:

If second curve is $[P_2 \quad P_3 \quad uT_2 \quad T_3]$ we have G_1, and if $u = 1$ we have C_1.

Bezier curves:

Defined by n + 1 control points. Endpoints are interpolated, inner ones are approximated.

Blending functions are the **Bernstein polynomials**.

Entirely contained within the convex hull of its control points.

The **de Casteljau** algorithm allows subdivision of a Bezier curve into two parts at an arbitrary parameter value t between 0 and 1.

Catmull-Rom Splines:

Piecewise cubic curve with tangent direction at each point parallel to the line segment connecting the two neighboring points. Requires no control points. C_1. Local control.

All the above cubic curves exhibit **local control**. Alternative for **global control**: **cubic spline**.

Cubic splines maintain C^2 everywhere. No geometrical constraints.

Cubic splines with zero second derivatives at the ends of the curve are called **natural cubic splines**.

B-splines:

Requires no control points. C^2. In exchange, gives up local-only (it's global instead). Stands for **basis spline**.

Uniform B-splines:

Points where they join together are called **knots**. The ends count too. A B-spline is called **uniform** if the knots are spaced at equal parameter values along the entire curve.

Nonuniform B-splines:

Cox-de Boor algorithm used to create blending functions.

NURBS:

Nonuniform **rational** B-splines: extend to homogeneous coordinates. Assign a weight w_i to each control point and express as $\langle w_i x_i, w_i y_i, w_i z_i, w_i \rangle$. Points on the curve are now a ratio of two polynomials (hence rational).

Weight influence is local. NURBS are invariant to homogeneous projection transformation. NURBS can represent any of the other curves, and can represent conic sections exactly.

Curvature and torsion:

Curvature κ is the magnitude of the rate at which the unit tangent vector changes per distance along the curve:

$$\kappa(t) = \left\| \frac{d}{ds} \hat{T}(t) \right\| = \frac{\left\| \frac{d}{dt} \hat{T}(t) \right\|}{\frac{ds}{dt}}$$

Where ds is:

$$ds = \left\| \frac{d}{dt} P(t) \right\| dt$$

$$\hat{T}(t) = \frac{T(t)}{\|T(t)\|} = \frac{\frac{d}{dt} P(t)}{\left\| \frac{d}{dt} P(t) \right\|} = \frac{\frac{d}{dt} P(t)}{\frac{ds}{dt}}$$

$$\kappa(t) = \frac{\|P'(t) \times P''(t)\|}{\|P'(t)\|^3}$$

$$\hat{N}(t) = \frac{\frac{d}{dt}\hat{T}(t)}{\left\|\frac{d}{dt}\hat{T}(t)\right\|}$$

Radius of curvature:

$$\rho(t) = \frac{1}{\kappa(t)}$$

The radius of curvature at a point P corresponds to the radius of a circle that is tangent to the curve at P and lies in the plane determined by the directions $\hat{T}$ and $\hat{N}$. This is called the **osculating plane**, and the circle is called the **osculating circle**.

$$a_T = \frac{d}{dt}v(t)$$

$$a_N = \frac{[v(t)]^2}{\rho(t)}$$

We can also complete a 3D orthonormal basis at P by defining the unit **binormal**:

$$\hat{B}(t) = \hat{T}(t) \times \hat{N}(t)$$

This is called the **Frenet frame**.

Frenet formulas:

$$\frac{d}{ds}\hat{T}(t) = \kappa(t)\hat{N}(t)$$

$$\frac{d}{ds}\hat{N}(t) = \tau(t)\hat{B}(t) - \kappa(t)\hat{T}(t)$$

$$\frac{d}{ds}\hat{B}(t) = -\tau(t)\hat{N}(t)$$

$\tau(t)$ is called the **torsion** and pertains to the amount by which the Frenet frame twists about the tangent as it travels:

$$\tau(t) = -\hat{N}(t) \cdot \frac{d}{ds}\hat{B}(t)$$

For a planar curve, torsion is 0.

Collision detection:

Plane collisions: determine what point on the object would be in contact with the plane, then represent the object by that point in a ray-and-plane intersection calculation.

Collision of a sphere and a plane:

To check that the positive distance between the center of the sphere and the plane is r, we have:

$$L \cdot P = r$$

$$N \cdot P + d - r = 0$$

$$L' = \langle N, D - r \rangle$$

Now we suppose the sphere's center moves from P_1 at $t = 0$ to P_2 at $t = 1$.

$$P(t) = P_1 + tV$$

$$V = P_2 - P_1$$

Intersection occurs when:

$$L' \cdot P(t) = 0$$

$$L' \cdot (P_1 + tV) = 0$$

$$t = -\frac{L' \cdot P_1}{L' \cdot V}$$

Where t must be between 0 or 1. Alternatively we can query early for collision without calculating t by checking. We assume:

$$L \cdot P_1 \geq r$$

And then check:

$$L \cdot P_2 < r$$

Just like intersection between a ray and a plane (since that's what it is).

We can plug back in to get P. However this point is the sphere's center at collision time. The collision point itself is P shifted toward the plane by r, so:

$$C = P(t) - rN$$

Collision of a box and a plane:

Assume the box has edges R, S, and T.

$$r_{eff} = \frac{1}{2}(|R \cdot N| + |S \cdot N| + |T \cdot N|)$$

With respect to a plane with normal N.

Let the box position be:

$$Q(t) = Q_1 + tV$$

$$V = Q_2 - Q_1$$

To find an intersection with $L = \langle N, D \rangle$, we calculate the ray intersection as before:

$$t = -\frac{L' \cdot Q_1}{L' \cdot V}$$

$$L' = \langle N, D - r_{eff} \rangle$$

We assume:

$$L \cdot P_1 \geq r$$

And then check:

$$L \cdot P_2 < r$$

If there's a collision, we have Q_1, and now must determine what point or set of points intersected.

If all three quantities $|R \cdot N|, |S \cdot N|, |T \cdot N|$ are nonzero, no edge is parallel to L, so we hit at one of the box's vertices. We find the vertex of contact by taking the negative versions of the quantities:

$$C = Q(t) - \frac{1}{2}[sgn(R \cdot N)R + sgn(S \cdot N)S + sgn(T \cdot N)T]$$

If exactly one quantity is 0, that axis of the box is parallel to the plane, and collision must occur on an edge. The two endpoints of the colliding edge are given by modifying the previous equation so that both signs are chosen for the zero term. For instance, if $|T \cdot N| = 0$:

$$C_{1,2} = Q(t) - \frac{1}{2}[sgn(R \cdot N)R + sgn(S \cdot N)S \pm T]$$

If two quantities are 0, the collision occurs on a whole face, and the four points come from choosing both signs for both zero terms. For instance, if $|S \cdot N| = 0$ and $|T \cdot N| = 0$:

$$C_{1,2,3,4} = Q(t) - \frac{1}{2}[sgn(R \cdot N)R \pm S \pm T]$$

Kareem Omar
kareem_omar@naughtydog.com

Spatial partitioning:

If you have a plane $L = \langle N, D \rangle$ that partitions world geometry, an object with effective radius r_{eff} lies entire on the positive side of the plane if:

$$L \cdot P \geq r_{eff}$$

And entirely on the negative side of the plane if:

$$L \cdot P \leq -r_{eff}$$

General sphere collision:

A sphere of radius r can be collided with an arbitrary polygonal model by expanding the model by r, as follows:

- Determine whether the sphere's center intersects any faces after they've been moved outward by r.
- If not, determine whether the sphere's center intersects any cylinders of radius r corresponding to expanded edges of the model.
- If not, determine whether the sphere's center insects any of the spheres of radius r corresponding to the expanded vertices of the model.

We don't have to worry about ensuring we're checking the exteriors of the cylinders or vertex spheres because we would have first intersected the exterior.

Ray->face is ray triangle.

Ray->sphere is ray sphere.

Ray->cylinder is harder because the cylinder is at arbitrary orientation.

First check bounding spheres – moving sphere of radius r with model of radius R, check a ray with sphere of radius $R + r$.

Note that you don't have to check cylinders or vertex spheres for two faces sharing an edge with an exterior angle <= 180 degrees, as the faces will hit first.

Two faces meet at an exterior angle <= 180 degrees if:

$$[N_1 \times (E_2 - E_1)] \cdot N_2 \geq 0$$

Sliding an object that wanted to go from P_1 to P_2 but hit something at Q on a surface with normal N:

$$P_3 = P_2 - [(P_2 - Q) \cdot N]N$$

Don't forget to then check collisions between Q and P_3!

Use proper normal coordinates, not interpolated vertex normals from Q's barycentric coordinates.

Collision of two spheres:

Let P and Q be the centers of two spheres. We have:

$$V_P = P_2 - P_1$$

$$V_Q = Q_2 - Q_1$$

So the squared distance between them is:

$$d^2 = \|P(t) - Q(t)\|^2$$

For convenience we define:

$$A = P_1 - Q_1$$

$$B = V_P - V_Q$$

Quadratic formula and take the smaller root:

$$t = \frac{-(A \cdot B) - \sqrt{(A \cdot B)^2 - B^2\left[A^2 - \left(r_P + r_Q\right)^2\right]}}{B^2}$$

If t does not fall between 0 and 1, no collision occurs during the interval of interest.

You can also check whether a collision occurs: the time at which the squared distance is minimized can be found by setting the derivative to zero as follows:

$$2B^2 t + 2(A \cdot B) = 0$$

$$t = -\frac{A \cdot B}{B^2}$$

$$d^2 = A^2 - \frac{(A \cdot B)^2}{B^2}$$

If:

$$d^2 > \left(r_P + r_Q\right)^2$$

Then the two spheres never collide.

If they do collide, get P and Q, then:

$$C = P(t) + r_P N$$

With:

$$N = \frac{Q(t) - P(t)}{\|Q(t) - P(t)\|}$$

Physics:

Second-order ODEs:

$$x'' + ax' + bx = f$$

If f is 0, this is a **homogeneous** ODE.

Linear combinations of solutions are also solutions.

Convert to auxiliary equation:

$$x'' + ax' + bx = 0$$

Becomes:

$$r^2 + ar + b = 0$$

Solve to get two r roots, r_1 and r_2.

IF $r_1 \neq r_2$:

$$x(t) = Ae^{r_1 t} + Be^{r_2 t}$$

Where A and B are arbitrary constants (that will be used to satisfy initial conditions).

If $r_1 = r_2$:

$$x(t) = Ae^{rt} + Bte^{rt}$$

If the roots of the quadratic are complex, the equation above is still valid, but we could re-express with real-valued functions using the fact that:

$$e^{\alpha+\beta i} = e^{\alpha}\big(\cos(\beta) + isin(\beta)\big)$$

If a and b were real, the roots are complex conjugates, so we have:

$$r_1 = \alpha + \beta i$$

$$r_2 = \alpha - \beta i$$

We can now express our solution as:

$$x(t) = e^{\alpha t}(C_1 \cos(\beta t) + C_2 \sin(\beta t))$$

Where C_1 and C_2 are arbitrary constants.

Which is equivalent to:

$$x(t) = De^{\alpha t}\sin(\beta t + \delta)$$

Where D and δ are arbitrary constants.

Nonhomogeneous:

If $f(t)$ is nonzero the solution takes the form:

$$x(t) = g(t) + p(t)$$

Where $g(t)$ is the homogeneous solution and p is a **particular solution**.

One method to find the particular solution is **undetermined coefficients**, where we guess at the structure. For example:

$$x'' - 5x' + 6x = 12t - 4$$

The general solution first:

$$r^2 - 5r + 6 = 0$$

$$(r-3)(r-2) = 0$$

$$r_1 = 2$$

$$r_2 = 3$$

$$g(t) = Ae^{2t} + Be^{3t}$$

We guess:

$$p(t) = Dt^2 + Et + F$$

Plug it into the ODE:

$$(Dt^2 + Et + F)'' - 5(Dt^2 + Et + F)' + 6(Dt^2 + Et + F) = 12t - 4$$

$$2D - 5(2Dt + E) + 6Dt^2 + 6Et + 6F = 12t - 4$$

$$2D - 10Dt - 5E + 6Dt^2 + 6Et + 6F = 12t - 4$$

$$6D = 0$$

$$D = 0$$

$$-10D + 6E = 12$$

$$E = 2$$

$$2D - 5E + 6F = -4$$

$$F = 1$$

Thus:

$$p(t) = 2t + 1$$

$$x(t) = Ae^{2t} + Be^{3t} + 2t + 1$$

Projectile motion:

Maximum height:

$$\dot{z} = v_z - gt = 0$$

$$v_z = gt$$

Time of maximum height:

$$t = \frac{v_z}{g}$$

Maximum height itself:

$$z = z_0 + v_z t - \frac{1}{2}gt^2$$

$$h = z_0 + v_z\left(\frac{v_z}{g}\right) - \frac{1}{2}g\left(\frac{v_z}{g}\right)^2$$

$$h = z_0 + \frac{v_z^2}{g} - \frac{1}{2}\frac{v_z^2}{g}$$

$$h = z_0 + \frac{1}{2}\frac{v_z^2}{g}$$

$$h = z_0 + \frac{v_z^2}{2g}$$

Range:

$$v_z t - \frac{1}{2}gt^2 = 0$$

$$t\left(v_z - \frac{1}{2}gt\right) = 0$$

$$\frac{1}{2}gt = v_z$$

$$t = \frac{2v_z}{g}$$

$$r = v_x t$$

$$r = \frac{2v_x v_z}{g}$$

Example: projectile launched at 30 m/s at 40 degrees above the horizontal. How far does it go?

$$v_x = 30\cos(40°) = 23$$

$$v_z = 30\sin(40°) = 19.3$$

$$r = 90.4\,m$$

Resisted motion:

$$x(t) = x_0 + \frac{g}{k}t + \frac{kv_0 - g}{k^2}\left(1 - e^{-kt}\right)$$

$$v(t) = \dot{x}(t) = \frac{g}{k} + \left(v_0 - \frac{g}{k}\right)e^{-kt}$$

Terminal velocity:

$$v_T = \frac{g}{k}$$

Friction:

$mgsin(\theta)$ down the incline

$$N = mgcos(\theta)$$

$$F_{fric} = \mu N = \mu mgcos(\theta)$$

$$F_{incline} = mgsin(\theta) - \mu mgcos(\theta)$$

$$a_{incline} = gsin(\theta) - \mu gcos(\theta)$$

Example: at what angle does a block with static friction coefficient $\mu_s = 0.5$ start moving?

$$mgsin(\theta) = 0.5mgcos(\theta)$$

$$\sin(\theta) = 0.5\cos(\theta)$$

$$\tan(\theta) = \frac{1}{2}$$

$$\theta = \arctan\left(\frac{1}{2}\right)$$

$$\theta = 26.6°$$

Rotational physics / rotating physics:

$$\omega(t) = \dot{\theta}(t) = \frac{d}{dt}\theta(t)$$

$$\boldsymbol{\omega}(t) = \omega(t)\boldsymbol{A} = \dot{\theta}(t)\boldsymbol{A}$$

$$v(t) = |\omega(t)r|$$

$$\boldsymbol{v}(t) = \boldsymbol{\omega}(t) \times \boldsymbol{r}(t)$$

Centrifugal force:

$$a(t) = \dot{v}(t) = \dot{\omega}(t) \times r(t) + \omega(t) \times \dot{r}(t)$$

$$a(t) = \dot{\omega}(t) \times r(t) + \omega(t) \times v(t)$$

$$a(t) = \dot{\omega}(t) \times r(t) + \omega(t) \times \big(\omega(t) \times r(t)\big)$$

The first term disappears for constant angular velocity, but the second term is always present. A pseudo force opposites this acceleration:

$$F_{centrifugal} = -m(\omega(t) \times [\omega(t) \times r(t)])$$

In the case that ω and r are perpendicular, we have:

$$F_{centrifugal} = m\omega^2 r = \frac{mv^2}{r}$$

Positions and its derivatives are true vectors, also known as polar vectors or contravariant vectors.

Angular velocities, magnetic fields, and surface normals are represented by pseudovectors, also known as axial vectors, covariant vectors, bivectors or 2-blades.

Doing two wedge products in a row in Grassman algebra, i.e. u ^ v ^ w = $u \cdot (b \times c)$, give a pseudoscalar (trivector, 3-blade) which is the signed volume of the parallelepiped.

Coriolis force:

Suppose a particle is rotating but also moving:

$$v(t) = \omega(t) \times r(t) + v_r(t)$$

But that means v_r is rotating too, so a stationary observer sees acceleration:

$$a_f(t) = \omega(t) \times v_r(t) + a_r(t)$$

Where $a_r(t)$ is the acceleration of the particle in the rotating reference frame.

Therefore:

$$a(t) = \dot{v}(t) = \dot{\omega}(t) \times r(t) + \omega(t) \times \dot{r}(t) + a_f(t)$$

$$= \dot{\omega}(t) \times r(t) + \omega(t) \times \dot{r}(t) + \omega(t) \times v_r(t) + a_r(t)$$

$$= \dot{\omega}(t) \times r(t) + \omega(t) \times [\omega(t) \times r(t)] + 2\omega(t) \times v_r(t) + a_r(t)$$

New term is Coriolis force:

$$F_{Coriolis} = -2m\omega(t) \times v_r(t)$$

Bends a moving particle in the direction of velocity cross rotation axis.

Rigid bodies:

Total mass M of system of particles:

$$M = \sum_k m_k$$

Center of mass:

$$C = \frac{1}{M} \sum_k m_k \boldsymbol{r}_k$$

Solid object:

$$M = \int_V dm(\boldsymbol{r}) = \int_V \rho(\boldsymbol{r})\, dV$$

Center of mass:

$$C = \frac{1}{M}\int_V \boldsymbol{r}\rho(\boldsymbol{r})\, dV$$

Angular momentum:

$$L(t) = r(t) \times p(t)$$

Where linear momentum:

$$p(t) = mv(t)$$

$$\dot{L}(t) = \dot{r}(t) \times p(t) + r(t) \times \dot{p}(t)$$

$$\dot{L}(t) = mv(t) \times v(t) + r(t) \times \dot{p}(t)$$

$$\dot{L}(t) = r(t) \times \dot{p}(t)$$

$$\dot{L}(t) = r(t) \times m\dot{v}(t)$$

$$\dot{L}(t) = r(t) \times F(t)$$

Torque:

$$\tau(t) = r(t) \times F(t)$$

$$\dot{L}(t) = \tau(t)$$

Angular momentum of a rigid body:

$$L(t) = I\omega(t)$$

$$\dot{L}(t) = \tau(t) = I\alpha(t)$$

Inertia tensor can be represented as a 3x3 matrix, where the diagonal terms are called the **moments of inertia** and the off-diagonals are the **products of inertia**.

Inertia tensors are sums, so you can find the inertia tensor of a complex object by summing the inertia tensors of its parts.

Principal axes of inertia:

Because the angular momentum and angular velocity are related by a multiplication by the inertia tensor, they are *parallel* when $\omega(t)$ is an eigenvector of the inertia tensor. Since the inertia tensor is a symmetric matrix, it has three real eigenvalues, and the associated eigenvectors are orthogonal! The eigenvalues are called the **principal moments of inertia**, and the associated eigenvectors are called the **principal axes of inertia.** If a rigid body is rotating about one of its principal axes, its angular momentum is just a scalar multiple (the associated eigenvalue) of the angular velocity.

If the inertia tensor is a diagonal matrix (sphere, cylinder), the principal moments of inertia are just the diagonal entries, and the principal axes are its x, y, and z axes.

If a rigid body is *not* rotating about a principal axis, the angular velocity and angular momentum are *not* parallel. Then the angular momentum rotates about the rotation axis at the rate:

$$\omega(t) \times L(t) \neq 0$$

The resulting angular acceleration changes the axis of rotation, an effect called **precession**. Since:

$$\dot{L}(t) = I\alpha(t)$$

The angular acceleration caused is:

$$\alpha(t) = I^{-1}\dot{L}(t) = I^{-1}[\omega(t) \times L(t)] \text{ (due to precession, not generally)}$$

To keep the axes in the same place we therefore must apply the opposite torque, so our final result is:

$$\tau(t) - \omega(t) \times L(t) = I\alpha(t) = I\ddot{\theta}(t)$$

Oscillatory motion:

Hooke's Law:

$$F = -kz$$

$$m\ddot{z} = -kz(t)$$

$$\omega = \sqrt{\frac{k}{m}}$$

$$f = \frac{\omega}{2\pi} = \frac{1}{2\pi}\sqrt{\frac{k}{m}}$$

$$P = \frac{1}{f} = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}}$$

$$z(t) = Csin(\omega t + \delta)$$

Or with gravity:

$$z(t) = Csin(\omega t + \delta) - \frac{mg}{k}$$

Pendulum motion:

$$\alpha(t) = -\frac{mgL}{I}\sin\big(\theta(t)\big)$$

Where I is the moment of inertia. If the puck is a point mass:

$$I = mL^2$$

And thus:

$$\alpha(t) = -\frac{g}{L}\sin\big(\theta(t)\big)$$

Can't be solved analytically, but if we use $\sin(\theta) \approx \theta$:

$$\alpha(t) = -\frac{mgL}{I}\theta(t)$$

$$\theta(t) = Asin(\omega t + \delta)$$

$$\omega = \sqrt{\frac{mgL}{I}}$$

$$P = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{I}{mgL}}$$

For the point mass:

$$\omega = \sqrt{\frac{g}{L}}$$

$$P = 2\pi\sqrt{\frac{L}{g}}$$

Wave equation:

$$\frac{\partial^2 z}{\partial t^2} = c^2\left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2}\right)$$

With damping term:

$$\frac{\partial^2 z}{\partial t^2} = c^2\left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2}\right) - \mu\frac{\partial z}{\partial t}$$

Fast sin and cos:

Lookup table of 256 entries between 0 and 2pi, then:

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

LU Decomp:

Doolittle's or Crout's method

Can be used to repeatedly solve linear systems with the same coefficient matrix M.

Using LU decomp to solve a linear system is approximately $\frac{2}{3}n^3$ flops.

Error reduction:

$$M(x+\Delta x) = r + \Delta r$$

$$Mx + M\Delta x = r + \Delta r$$

Subtract the original system, $Mx = r$:

$$M\Delta x = \Delta r$$

$$M\Delta x = Mx + M\Delta x - r$$

$$M\Delta x = Mx_0 - r$$

The entire right side is known, so we can solve for Δx and subtract that from our original solution to iterate our answer.

Tridiagonal systems:

M is **diagonally dominant** if the absolute value of the diagonal element is greater than the sum of the absolute values of the terms to its left and right, for all rows.

Tridiagonal linear systems can be solved efficiently.

If M is diagonally dominant, a solution **must** exist and M **must** be invertible (nonsingular).

Jacobi method can be used to diagonalize a matrix.

Euler's method:

$$y_{i+1} = y_i + hf(x_i, y_i)$$

Permutation: SEPA / Knuth's Algorithm L. Heap's algorithm uses fewer swaps.

PRNG: random number generator: Mersenne twister, mother-of-all, xorshift

Kareem Omar
kareem_omar@naughtydog.com

Memory model

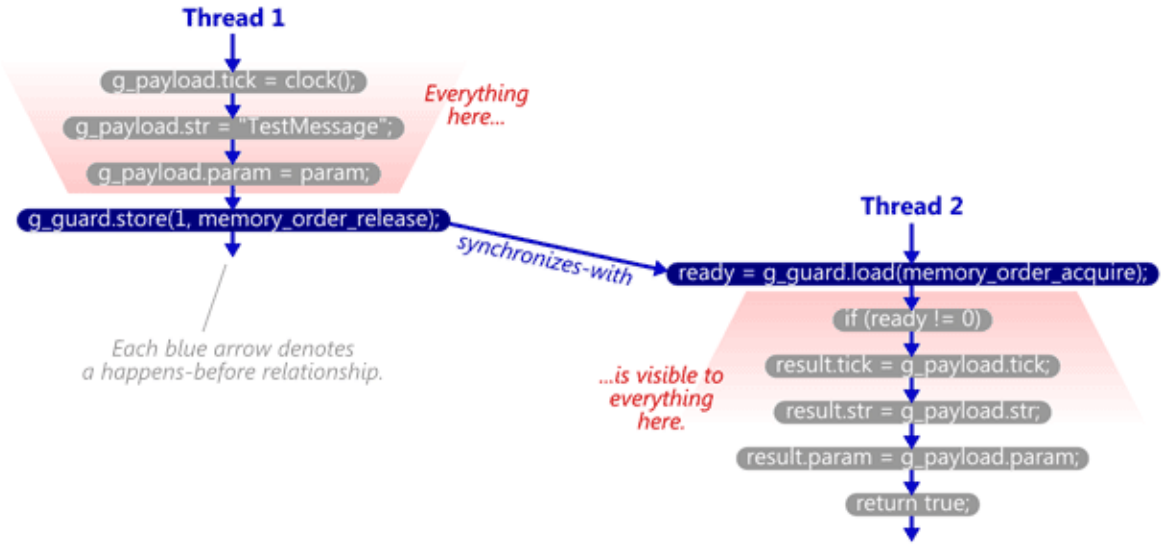
Compiler, processor architecture, or memory architecture can *all* reorder your stuff and ruin your day. Fortunately, these are conceptually *equivalent*, and considering the consequences of reordering at *any* level can be accomplished by considering any other level of your choice, so, for example, you can reason about reorderings by considering only reordering of source code – with one, horrifically subtle caveat: you must consider whether your reordering can cause correctness problems if the thing that gets reordered later is just a little *late*, or whether it *never* happens. This can differ between architecture and compilers, as will be explained a bit later.

In x86 (CPU + memory arch), of the four possible reorderings, for most instructions, the only reordering that happens is that stores can be moved after unrelated loads, or equivalently, loads can be moved before unrelated stores. This is called StoreLoad reordering. (In general, the convention is, XY reordering means that X can be moved after unrelated Y.) This means x86 has TSO (total store order). Again, this is not taking into account compiler reordering.

Load-Acquire semantic. Prevents even unrelated loads and stores from moving before it. (i.e. prevents LoadLoad and LoadStore reordering, which is true for ALL normal loads on x86). Consumer behavior. Likely implemented by flushing inval Q. Order: read a control value with load-acquire semantics, then read your data. Or, if using explicit fences, read a control value, LoadLoad and LoadStore barrier, then read your data. Makes sure anything that's been invalidated is re-read from scratch. All writes from anyone who released the same variable are visible locally, which is the desired state for a consumer waiting for permission to begin operating on some data.

Store-Release semantic. Prevents even unrelated loads and stores from moving after it (i.e. prevents LoadStore and StoreStore reordering, which is true for ALL normal stores on x86). Producer behavior. Likely implemented by flushing store Q. Order: write the data, write the control value with store-release semantics. Or, if using explicit fences, write the data, LoadStore and StoreStore barrier, then write the control value. Makes sure the data we just wrote is globally visible. All writes released will be visible to anyone who acquires.

In the case of explicit fences, an acquire fence can only promote a *preceding* read to a load-acquire, and a release fence can only promote a *following* write to a store-release. This was indicated in the order of operations for explicit fences I specified above; this explains why that order must be used.



Because StoreLoad is still allowed on x86, sequential consistency is not achieved. (Two threads can be in compliance with all the above yet disagree about the global order of memory events). SC is rarely needed, however.

Time to explain the horrifically subtle caveat from earlier. Example:

Time	Thread 0 Actions	Thread 1 Actions
t = 0	Lock Mutex A	
t = 1		Lock Mutex B
…		Release Mutex B
	Lock Mutex B	
	Release Mutex A	
		Lock Mutex A

In general, there is the possibility of deadlock if two threads try to grab two mutexes in opposite order: if 0 has A and is waiting for B, and 1 has B and is waiting for A, they'll be stuck forever: deadlock. However, this example should be *safe*. Thread 0 grabs both locks, but thread 1 never does.

But what if Thread 1's Release Mutex B and Lock Mutex A get reordered?? Then we'd have exactly the situation described above, in which two threads try to acquire two mutexes in opposite order, and we could get deadlock. What's scarier is, that's caused by moving a Release after a Load… that's StoreLoad reordering, which is allowed by x86! Does this mean we really do need SC everywhere, not just the default release and acquire semantics, to prevent this kind of reordering introducing deadlock?

Fortunately not. In this case, it *matters* whether it's the compiler or the architecture doing the reordering. The barriers in this example are sound; this is not an issue of data correctness. It's an issue of deadlock when code gets stuck waiting for a lock. Waiting for a lock is not a load or a store, it's a *loop* (that involves repeatedly loading – like a flag value, if it's a spinlock). What we actually have for our thread 1's release B and then lock A is:

```
StoreRelease(B, UNLOCKED);

while (LoadAcquire(A) == LOCKED) {}
```

Okay, this is a store followed by between 1 and infinity loads. A *compiler* could indeed reorder this to:

```
while (LoadAcquire(A) == LOCKED) {}

StoreRelease(B, UNLOCKED);
```

Which could introduce deadlock. So, indeed, in this case we MUST prevent compiler reordering.

But should we be concerned about x86 reordering in this case? Will we have to put mfence instructions in to prevent it?

Fortunately not. In the compiler reorder above, the while loop would spin *forever*, and the store would never execute, as it's after the load(s), and the architecture cannot move it before (that's LoadStore reordering, not permitted on x86).

If we prevent compiler reordering, so the architecture sees our original desired order:

```
StoreRelease(B, UNLOCKED);

while (LoadAcquire(A) == LOCKED) {}
```

Then the architecture sees a store followed by some loads. Yes, the architecture allows StoreLoad reordering, so it CAN delay the store… **but unlike the compiler it will not delay it forever!** The while loop may begin executing and load a few times before the store commits, but the store will *eventually* commit. The architecture is not in the business of reordering a store past an infinite number of loads! It would not want to, but also it physically cannot, as its reorder buffer is not infinitely large. So after some (small) number of the loads, it WILL commit the store, which will become globally visible. You have nothing to worry about in this case and do NOT require SC (and thus do not require an explicit mfence). You DO, however, need to ensure your *compiler* does not reorder you in this case.

You can accomplish this with _ReadWriteBarrier, a compiler-only reorder barrier. However, this is too coarse – a complete barrier. It's much preferred to use something fine-grained (C++11 std::atomics!) to tell the compiler your exact intentions: the kind of semantics (release, acquire, or SC) you want to attach to a *particular* variable.

An example of the rare case in which you do actually need SC: Dekker's algorithm. Assume we enter this with global values X and Y both equal to 0.

Time	Thread 0 Actions	Thread 1 Actions
t = 0	Store 1 to X	Store 1 to Y
t = 1	Load Y into R1	Load X into R2

With a little work, this can be used for mutual exclusion. Both loads occur after a store. Therefore, by the time a load is hit, at least one store has occurred. Therefore, by the time *both* loads are hit, at least one store has occurred. Therefore, at least one of the loads will be 1. One or both reads will be 1. It is not possible for both reads to be 0. Therefore this can be used for mutual exclusion: if I'm 0, enter the critical section. (Obviously it would need work to deal with the case where you get a 1 and do not enter the critical section, but you get the basic idea.)

Locally, things would look in-order to the core that issued the stores and loads – this is always true on x86. A core sees its own stores and loads in-order.

However, what if threads 0 and 1 are running on different cores? X and Y are unrelated, and the order of each of these threads is a Store followed by a Load. So x86 could reorder both threads to do their loads first, which, combined with both threads being on different cores and executing close enough in time that the loads both complete before either store, means you COULD have both loads be 0.

In this case, thread 0's core would see X get set to 1, *then* see Y get set to 1 (by the other core).

Thread 1's core would see Y get set to 1, *then* see X get set to 1 (by the other core).

The two cores disagree about the global order of events.

This is what is meant by sequential consistency! We need SC for Dekker's algorithm to function correctly.

x86 provides some instructions for this stuff:

LFENCE is a LoadLoad fence and a LoadStore fence (usually you get these anyway; see exceptions below).

SFENCE is a StoreStore fence. (usually you get this anyway; see exceptions below).

MFENCE is all 4 fences, most importantly a StoreLoad fence. Neither LFENCE, SFENCE, nor LFENCE + SFENCE prevents a store followed by a load from being reordered. MFENCE does.

MFENCE only prevents reordering of the global visibility of our own stores and loads. It can't stop a store from another core from becoming visible to us after our loads but before our stores (ruining a read-modify-write like an increment of a global variable). That requires an atomic read-modify-write bus cycle, which is what locked instructions are for.

Single x86 instructions ARE atomic without a LOCK on single-threaded systems since interrupts and context-switches, the only things that could break atomicity, never happen in the middle of an instruction, only between instructions. However, they are NOT atomic without the LOCK prefix on multicore systems, and they are not relative to hardware observers.

Now for the exceptions to x86's strongly ordered memory model. The `rep stos*` and `rep movs*` families of string manipulators are StoreStore reordered, but only with respect to their own operations, so it's not a big problem; basically, don't check to see if these operations are done by polling the last element they are supposed to write out, as they may be visible to you out of order. However, a separate "done" flag set after the string manipulator and checked with normal loads and stores (release-acquire for free!) is sufficient.

The other family is `movnt*`, the non-temporal or streaming loads and stores. The loads are unaffected, but the stores are StoreStore reorderable. They're invaluable for speeding up writes of data you know you won't touch again in the same algorithm, but that does mean you need an `sfence` at the very end.

Assuming writes to normal WB memory, then, lfence is never required for anything, sfence is only required after movnt, don't use the output of a string manipulator to determine when it's done, and mfence is only required for sequential consistency.

The consume flag is subtle, like, Herb Sutter is afraid of it, and if Herb Sutter is afraid of a thing I should be as well. It has value in enforcing physical dependency, like, anything literally connected to this address will not be reordered with this load (even from other threads/cores), but every architecture ever except for the DEC Alpha does this automatically. It would be crazy if they didn't give this basic guarantee, that writes/reads to the *same* memory are not reordered. All the other discussion has been for writes/reads to different (perhaps conceptually related but not literally overlapping) memory.

Compare and Swap (CAS): operates on the cache layer, not main memory. CompareAndExchange(addr, expected, newValue). Returns the previous value so you can check (ret == expected ? success : failure) to see whether it succeeded. x86: lock cmpxchg [dest], newValue, with expected value in eax.

Align shared data/flags to cache line to prevent thrashing – in fact, align to 128 due to hardware prefetching.

sig_atomic_t is just a typedef to something that can be read or written in 1 cycle; not sufficient for RMW, but its existence means that on your implementation, variables of that type can be read or written atomically.

The C++11 atomics default to SC-DRF too but allow you to specify relaxed memory models:

```
x.store(1, std::memory_order_release);
x.load(std::memory_order_acquire);
```

Which serve as just the right amount of compiler barrier (one-directional, as desired!), do not inhibit constant propagation and other harmless optimizations across them, and generate optimal codegen for x86 (normal loads and stores)! As long as you ensure you don't need sequential consistency. And if you do, easy, just replace with std::memory_order_seq_cst!

xchg instruction with a memory operand implies LOCK prefix.

Kareem Omar
kareem_omar@naughtydog.com

Cache coherency protocols

AMD uses MOESI. Skylake uses MESIF.

MSI: modified, shared, invalid.

Modified: this cache line has been modified by this core. It now differs from main memory and must be written back when evicted.

Shared: this cache line matches main memory and is present in 1 or more cores' caches. It can be freely evicted without writeback.

Invalid: this cache line is either not present or has been invalidated by a bus request. It must be fetched from memory or another cache if requested.

Disadvantage 1: for another core to read/write a dirty line, the line must first be written out (change to S), then shared. This is solved by allowing multiple caches to have an (equivalent) copy of a modified element instead of always writing back. The responsible cache is called OWNED. This is solved by O state. Shared now means multiple caches have it, not the owner, but could be either clean *or* dirty. Owned means dirty, owner. Modified means dirty, sole user. MSI -> MOSI

Disadvantage 2: to read a clean line no one else has (common! Single-threaded app), MSI must ask all other cores first, consuming bus traffic. This is solved by E state. Shared means *clean*, not sole user. Exclusive means clean, sole user. Modified means dirty, sole user. MSI -> MESI

Disadvantage 3: to read a clean shared line, MSI must either get bombarded with answers from all cores which have it, or go to main memory. This is solved by F state. (Forward). Shared means multiple caches have it, not the owner, but could be either clean or dirty. Like the Owned in MOESI, Forward indicates the owner of a cache line, but unlike MOESI, it indicates the owner of a *clean* cache line. MSI -> MSIF

MOESI, lacking the ability to mark the owner of a shared clean line, cannot efficiently share clean lines, which will come from memory.

MESIF, lacking the ability to mark the owner of a shared dirty line, cannot efficiently share dirty lines, which will come from memory.

You could do MOESIF, in theory, but this is not done in current CPUs.

Unaligned locked ops are a problem: we need other cores to see modifications to two cache lines happen as a single atomic operation. This may require actually storing to DRAM, and taking a bus lock. (AMD's optimization manual says this is what happens on their CPUs when a cache-lock isn't sufficient.)

Games are soft real-time interactive agent-based computer simulations.

Kareem Omar
kareem_omar@naughtydog.com

Questions to be able to **derive** + answer **fully from memory**

(note that derivations are not always given, sometimes only final answers)

Question: How many floating point numbers are in the interval [0, 1]?

0 is represented by a 0 exponent and a 0 mantissa (0.0×2^0) which is the bit pattern all 0s.

1 is represented by a 127 exponent and a 0 mantissa (1.0×2^0) which is the bit pattern 1111111 followed by 23 0s

So the answer is just 0b11111110000000000000000000000000 + 1

Which is 0x3F800001

Which is 1065353217

Okay, what about normals only?

FLT_MIN is represented by a 1 exponent and a 0 mantissa (1.0×2^{-126}) which is the bit pattern 1 followed by 23 0s.

1 is represented by a 127 exponent and a 0 mantissa (1.0×2^0) which is the bit pattern 1111111 followed by 23 0s

Answer is (127 << 23) – (1 << 23) + 1 = 1056964609

Plus 1 if we count 0:

1056964610

Question: Find a line through a point P parallel to the plane Ax+By+Cz+D=0.

There isn't just one; any line perpendicular to the plane's normal vector N that passes through point P will work. That's a plane. So our new plane has points P and Q, and our line connects P and Q.

Normal vector:

$$N = \frac{(A, B, C)}{|(A, B, C)|}$$

New plane:

$$Ax + By + Cz + D = 0$$

$$D = -N \cdot P$$

$$\boxed{Ax + By + Cz - AP_x - BP_y - CP_z = 0}$$

$$\boxed{A(x - P_x) + B(y - P_y) + C(z - P_z) = 0}$$

To recap, the first point of our line is P. The new point is:

$$Q = (x, y, z)$$

For any Q that satisfies the above equation, $Q \neq P$.

Note that we have 3 variables and 1 equation for a total of 2 free variables. This is sufficient to define an angle and a radius within the plane, completely covering all possible solution points. Makes sense!

Question: what is the value of the float with bit pattern 0x4048F5C3?

NOTE: start counting from the right!!

In binary this is 01000000010010001111010111000011

Sign bit is 0, so this is a positive number.

Exponent is 128 – 127 = 1 so this number is in the window that starts at 2^1, so [2, 4).

And from the mantissa we are 0b10010001111010111000011 / 2^23 of the way through the window. The window is 2 wide. This means we are at the base of the window + 2 * 0b10010001111010111000011 / 2^23 = 2 + 2 * 0b10010001111010111000011 / 2^23 = 3.140000104904175

To express this more generally:

$$bottom\ of\ window + (-1)^S \frac{M}{2^{23}}(window\ size)$$

$$2^{E-127} + (-1)^S \frac{M}{2^{23}}(2^{E-127})$$

More mathematically:

For E between 1 and 254 inclusive:

$$(-1)^S * \left((1 \ll 23) + \mathrm{M}\right) * 2^{\mathrm{E}-127-23}$$

$$(-1)^S * (0x800000 + \mathrm{M}) * 2^{\mathrm{E}-127-23}$$

$$(-1)^S * (8388608 + \mathrm{M}) * 2^{\mathrm{E}-127-23}$$

For E = 0, denormal number:

$$(-1)^S * \mathrm{M} * 2^{-126-23}$$

$$(-1)^S * \mathrm{M} * 2^{-149}$$

For E = 255, infinity or NaN:

$$If\ M = 0, infinity$$

$$If\ M \neq 0, NaN$$

Question: what is the bit representation of the float nearest to 3.14?

Positive so S = 0.

The next lowest power of 2 is 2, i.e. 2^1, so the exponent is 1 + 127 = 128.

E = 128.

We're in the window [2, 4). We're (3.14 – 2) / (4 – 2) of the way through the window, which is 0.57. Multiply that by 2^23 and round to nearest int to get the mantissa:

M = 0x48f5c3

To express this more generally:

$$e = \lfloor \log_2 x \rfloor$$

$$E = e + 127$$

We either pick the mantissa that selects our fractional progress into our window, or generate the full number and subtract off the leading 1 bit:

$$M = round\left(\frac{x - 2^e}{2^e} * 2^{23}\right) = round\left(\frac{x}{2^e}2^{23} - 2^{23}\right)$$

$$I = (E \ll 23) + M$$

Discuss DCMs/quats to axis-angle and rotation about an arbitrary axis (i.e. the Rodrigues Formulation, i.e. the exp map):

DCMs are Lie group $SO(3)$

Quaternions are Lie group $SU(2) = Spin(3)$

Axis-angle is Lie algebra $so(3)$ = su(2)

exp: $so(3) \rightarrow SO(3)$ (axis-angle to DCM/quat)

log: $SO(3) \rightarrow so(3)$ (DCM/quat to axis-angle)

$$P' = Pcos(\theta) + (A \times P)\sin(\theta) + A(A \cdot P)(1 - \cos(\theta))$$

$$R = \exp(\theta K) = I + \sin(\theta)\,K + (1 - \cos(\theta))K^2$$

Log map for DCMs:

$$\theta = \arccos\left(\frac{Tr(R) - 1}{2}\right)$$

$$\omega = \frac{1}{2\sin(\theta)}\begin{bmatrix} R(3,2) - R(2,3) \\ R(1,3) - R(3,1) \\ R(2,1) - R(1,2) \end{bmatrix}$$

Exp map for quats:

$$q = \left(\cos\left(\frac{\theta}{2}\right), \omega \sin\left(\frac{\theta}{2}\right)\right)$$

Log map for quats:

$$\theta = 2\arccos(s) = 2atan2(|v|, s)$$

$$\omega = \frac{v}{\sin\left(\frac{\theta}{2}\right)}$$

Matrix stuff:

The eigenvalues of a real symmetric matrix are real.

If $A^{-1}MA$ is diagonal, A's columns are the eigenvectors of M, and the diagonal entries are the corresponding eigenvalues.

If a matrix is normal ($A^*A = AA^*$) (conjugate transpose), including real symmetric and real orthogonal matrices, it has orthogonal eigenvectors for all *distinct* eigenvalues, so we can diagonalize it just by finding the eigenvalues and eigenvectors and placing the normalized eigenvectors in A. Then we have $A^{-1}MA = A^TMA = D$ and $M = ADA^{-1} = ADA^T$.

A symmetric matrix has n eigenvalues and there exist n linearly independent eigenvectors (because of orthogonality) *even if the eigenvalues are not distinct*.

What is the form of a 4x4 transform?

$$F = \begin{bmatrix} M & M & M & T_x \\ M & M & M & T_y \\ M & M & M & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} M & T \\ 0 & 1 \end{bmatrix}$$

What is the inverse of a 4x4 transform?

$$F^{-1} = \begin{bmatrix} M^{-1} & -M^{-1}T \\ 0 & 1 \end{bmatrix}$$

Derive a suitable transformation matrix for transforming covariant, such as normal, vectors.

$$G = (M^{-1})^T$$

What do you call vectors transformed with just M, in the ordinary way?

Contravariant.

Best way to propagate quats?

Adams-Bashforth-Moulton (ABM) adaptive variable-order linear multistep predictor-corrector solver.

Best way to apply a quat?

$$P' = 2(v \cdot P)v + (s^2 - v \cdot v)P + 2s(v \times P)$$

Kareem Omar
kareem_omar@naughtydog.com

What's the formula for SLERP?

$$q(t) = \frac{\sin\big(\theta(1-t)\big)}{\sin(\theta)} q_1 + \frac{\sin(\theta t)}{\sin(\theta)} q_2$$

$$\theta = \arccos(q_1 \cdot q_2)$$

What's the parametric equation of a line?

$$P(t) = (1-t)P_1 + tP_2$$

Derive the distance between a point and a line.

$$d = \sqrt{(Q-S)^2 - \frac{[(Q-S)\cdot V]^2}{V^2}}$$

Derive the distance between two lines, or at least the parameters t_1 and t_2 of closest approach.

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \frac{1}{(V_1 \cdot V_2)^2 - V_1^2 V_2^2} \begin{bmatrix} -V_2^2 & V_1 \cdot V_2 \\ -V_1 \cdot V_2 & V_1^2 \end{bmatrix} \begin{bmatrix} (S_2 - S_1) \cdot V_1 \\ (S_2 - S_1) \cdot V_2 \end{bmatrix}$$

Derive the distance between a point and a plane.

$$d = \frac{N \cdot Q + D}{\|N\|}$$

Derive the reflection of a point across an arbitrary plane L.

$$P' = P - 2\left(\frac{N \cdot Q + D}{\|N\|^2}\right) N$$

Derive the intersection of a line and a plane.

$$t = -\frac{N \cdot S + D}{N \cdot V} = -\frac{L \cdot S}{L \cdot V}$$

Derive the point of intersection of three planes.

$$Q = \begin{bmatrix} N_1 \\ N_2 \\ N_3 \end{bmatrix}^{-1} \begin{bmatrix} -D_1 \\ -D_2 \\ -D_3 \end{bmatrix}$$

Derive a suitable transformation matrix for transforming planes in homogeneous coordinates.

$$G = (F^{-1})^T L$$

Derive the equation for distance to the projection plane, e, in terms of horizontal field of view, α.

$$e = \frac{1}{\tan\left(\frac{\alpha}{2}\right)}$$

Derive the equation for vertical field of view, β, in terms of aspect ratio, a, and e.

$$\beta = 2\arctan\left(\frac{a}{e}\right)$$

Derive the 6 frustum planes in homogeneous coordinates.

Plane	$\langle \mathbf{N}, D \rangle$
Near	$\langle 0, 0, -1, -n \rangle$
Far	$\langle 0, 0, 1, f \rangle$
Left	$\left\langle \frac{e}{\sqrt{e^2+1}}, 0, -\frac{1}{\sqrt{e^2+1}}, 0 \right\rangle$
Right	$\left\langle -\frac{e}{\sqrt{e^2+1}}, 0, -\frac{1}{\sqrt{e^2+1}}, 0 \right\rangle$
Bottom	$\left\langle 0, \frac{e}{\sqrt{e^2+a^2}}, -\frac{a}{\sqrt{e^2+a^2}}, 0 \right\rangle$
Top	$\left\langle 0, -\frac{e}{\sqrt{e^2+a^2}}, -\frac{a}{\sqrt{e^2+a^2}}, 0 \right\rangle$

Table 5.1. View frustum plane vectors in OpenGL camera space in terms of the focal length e, the aspect ratio a, the near plane distance n, and the far plane distance f.

Derive projected point $\langle p, -e \rangle$ in the x-z plane for an edge $ax + bz = c$ and show that the reciprocal of the z-coordinate is correctly interpolated in a linear manner across the face of a triangle.

$$\frac{1}{z_3} = \frac{1}{z_1}(1-t) + \frac{1}{z_2}t$$

Derive the same interpolation for a vertex attribute b.

$$b_3 = z_3\left[\frac{b_1}{z_1}(1-t) + \frac{b_2}{z_2}t\right]$$

Given left edge $x = l$, right edge $x = r$, bottom edge $y = b$, and top edge $y = t$ of the rectangle carved out of the near plane (NOT the projection plane) by the view frustum, derive the perspective projection coordinates x', y', and z' for a point $\langle P_x, P_y, P_z, 1 \rangle$ and then the perspective projection matrix.

$$x' = \frac{2n}{r-l}\left(-\frac{P_x}{P_z}\right) - \frac{r+l}{r-l}$$

$$y' = \frac{2n}{t-b}\left(-\frac{P_y}{P_z}\right) - \frac{t+b}{t-b}$$

$$z' = -\frac{2nf}{f-n}\left(-\frac{1}{P_z}\right) + \frac{f+n}{f-n}$$

In normal coordinates, or:

$$P' = \langle -x'P_z, -y'P_z, -z'P_z, -P_z \rangle$$

In homogeneous coordinates.

Therefore:

$$M_{frustum} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Given left edge $x = l$, right edge $x = r$, bottom edge $y = b$, and top edge $y = t$ of the rectangle carved out of the near plane (NOT the projection plane) by the view frustum, derive the orthographic projection coordinates x', y', and z' for a point $\langle P_x, P_y, P_z, 1 \rangle$ and then the orthographic projection matrix.

$$x' = \frac{2}{r-l}x - \frac{r+l}{r-l}$$

$$y' = \frac{2}{t-b}y - \frac{t+b}{t-b}$$

$$z' = -\frac{2}{f-n}z - \frac{f+n}{f-n}$$

$$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Derive Newton's method.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Derive Newton's method for refining reciprocals, and the first guess requirement.

$$x_{n+1} = x_n(2 - rx_n)$$

$$0 < x_0 < \frac{2}{r}$$

Derive Newton's method for refining inverse square roots, and the first guess requirement.

$$x_{n+1} = \frac{1}{2}x_n(3 - rx_n^2)$$

$$0 < x_0 < \sqrt{\frac{3}{r}}$$

Intersect a ray with a triangle.

$$R = P - P_0$$

$$Q_1 = P_1 - P_0$$

$$Q_2 = P_2 - P_0$$

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \frac{1}{Q_1^2 Q_2^2 - (Q_1 \cdot Q_2)^2} \begin{bmatrix} Q_2^2 & -Q_1 \cdot Q_2 \\ -Q_1 \cdot Q_2 & Q_1^2 \end{bmatrix} \begin{bmatrix} R \cdot Q_1 \\ R \cdot Q_2 \end{bmatrix}$$

The point lies inside if all three weights are nonnegative, i.e. $w_1 \geq 0, w_2 \geq 0, w_1 + w_2 \leq 1$.

These weights can also be used for interpolation of vertex properties.

Intersect a ray with a box given by $\boldsymbol{x = 0, y = 0, z = 0, x = r_x, y = r_y, z = r_z}$.

Check which <= 3 planes require testing, find the intersection, and make sure the other two coordinates are in bounds.

Intersect a ray with a sphere.

$$a = V^2$$

$$b = 2(S \cdot V)$$

$$c = S^2 - r^2$$

$$t = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Intersect a ray with an ellipsoid.

$$x^2 + m^2y^2 + n^2z^2 = r^2$$

m is the ratio of the x semiaxis length to the y semiaxis length

n is the ratio of the x semiaxis length to the z semiaxis length

$$a = V_x^2 + m^2V_y^2 + n^2V_z^2$$

$$b = 2\left(S_xV_x + m^2S_yV_y + n^2S_zV_z\right)$$

$$c = S_x^2 + m^2S_y^2 + n^2S_z^2 - r^2$$

Intersect a ray with a cylinder.

$$x^2 + m^2y^2 = r^2$$

$$0 \le z \le h$$

$$a = V_x^2 + m^2V_y^2$$

$$b = 2\left(S_xV_x + m^2S_yV_y\right)$$

$$c = S_x^2 + m^2S_y^2 - r^2$$

Derive the reflection vector, $\boldsymbol{R}$.

$$R = 2(N \cdot L)N - L$$

Derive the refraction vector, $\boldsymbol{T}$.

$$T = \left(\frac{\eta_L}{\eta_T}(N \cdot L) - \sqrt{1 - \frac{\eta_L^2}{\eta_T^2}[1 - (N \cdot L)^2]}\right)N - \frac{\eta_L}{\eta_T}L$$

What's the point light equation?

$$C = \frac{1}{k_c + k_ld + k_qd^2}C_0$$

What's the spot light equation?

$$C = \frac{\max(-R \cdot L, 0)^p}{k_c + k_ld + k_qd^2}C_0$$

What's the ambient term?

$$K = DTA$$

What's the diffuse term?

$$K = DTC \max(N \cdot L, 0)$$

What's the specular term?

$$K = SGCmax(R \cdot V, 0)^m (N \cdot L > 0)$$

What's the emission term?

$$K = \mathcal{EM}$$

What's the approximated specular term?

$$H = \frac{L + V}{2}$$

$$K = SGCmax(N \cdot H, 0)^m (N \cdot L > 0)$$

What are the texture coordinate names?

$$s, t, p, q$$

If we sample a 2D texture with width w and height h, sample and BLERP the texture and determine the level-of-detail parameter, λ.

$$i = \left\lfloor ws - \frac{1}{2} \right\rfloor$$

$$j = \left\lfloor ht - \frac{1}{2} \right\rfloor$$

$$\alpha = frac\left(ws - \frac{1}{2}\right)$$

$$\beta = frac\left(ht - \frac{1}{2}\right)$$

$$\mathcal{T} = (1 - \alpha)(1 - \beta)T_{i,j} + \alpha(1 - \beta)T_{i+1,j} + (1 - \alpha)\beta T_{i,j+1} + \alpha\beta T_{i+1,j+1}$$

Let n and m be the base-2 logarithms of the width and height of the 2D texture map and $s(x, y)$ and $t(x, y)$ be functions that map viewport coordinates x and y to texture coordinates s and t. Then:

$$u(x, y) = 2^n s(x, y)$$

$$v(x,y) = 2^m t(x,y)$$

$$\rho_x = \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}$$

$$\rho_y = \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}$$

$$\lambda = \log_2\left[\max(\rho_x, \rho_y)\right]$$

Blend from levels $\lfloor\lambda\rfloor$ and $\lfloor\lambda\rfloor + 1$:

$$\mathcal{T} = \big(1 - frac(\lambda)\big)\mathcal{T}_1 + frac(\lambda)\mathcal{T}_2$$

What's Gouraud shading?

Only texture coordinates are per-pixel. Lighting is computed at the vertices and is interpolated.

What's Blinn-Phong shading?

Normals, L, and V are interpolated, and lighting is evaluated per-pixel. May want to renormalize the normal vectors to avoid noticeable specular darkening.

What is the unperturbed vector of a bump map?

$$\langle 0,0,1 \rangle$$

If we have a height map H, generate suitable tangent vectors S and T.

$$S = \langle 1,0, aH(i+1,j) - aH(i-1,j) \rangle$$

$$T = \langle 0,1, aH(i,j+1) - aH(i,j-1) \rangle$$

$$N = \frac{S \times T}{\|S \times T\|} = \frac{\langle -S_z, -T_z, 1 \rangle}{\sqrt{S_z^2 + T_z^2 + 1}}$$

Construct tangent (T, B, N) space such that z is the vertex normal, x is the s direction, and y is the t direction. In other words:

$$Q - P_0 = (s - s_0)T + (t - t_0)B$$

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{s_1 t_2 - s_2 t_1} \begin{bmatrix} t_2 & -t_1 \\ -s_2 & s_1 \end{bmatrix} \begin{bmatrix} (Q_1)_x & (Q_1)_y & (Q_1)_z \\ (Q_2)_x & (Q_2)_y & (Q_2)_z \end{bmatrix}$$

Then $[T \quad B \quad N]$ takes tangent space to object space. No need to store bitangent since it can be obtained from $N \times T$ up to handedness (so store that, possibly as the w-coordinate of T.

Summarize BRDF.

Bidirectional reflectance distribution functions. Radiance is:

$$C_R(V) = \varrho(V, L)C(N \cdot L)$$

$$\varrho(V, L) = kD + (1 - k)\varrho_s(V, L)$$

Cook-Torrance Illumination:

$$\varrho_s(V, L) = \mathcal{F}(V, L)\frac{D(V, L)G(V, L)}{\pi(N \cdot V)(N \cdot L)}$$

Where $\mathcal{F}$ is the Fresnel factor, D is the microfacet distribution function, and G is the geometrical attenuation factor.

Define light radiation terms.

Flux – energy emitted or received per unit time – W.

Flux Density - flux per unit area – W / m^2.

Emitted: radiosity

Incident: irradiance

Radiance – flux density per unit solid angle – W / m^2 / sr.

Spherical coordinates.

Polar angle φ (“phi”), azimuthal angle θ.

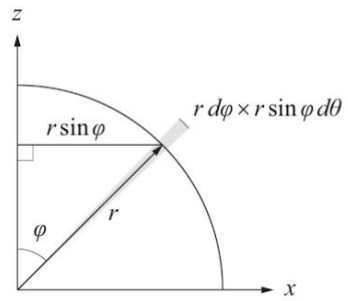


Figure 7.19. The differential surface area at the point $\langle r, \theta, \varphi \rangle$ on a sphere is equal to $r^2 \sin\varphi \, d\theta \, d\varphi$.

$$dA = r^2 \sin(\varphi)\, d\theta d\varphi$$

$$d\omega = \sin(\varphi)\, d\theta d\varphi$$

Kareem Omar
kareem_omar@naughtydog.com

Principal Component Analysis and constructing OBBs.

$$m = \frac{1}{N}\sum_{i=1}^{N} P_i$$

$$C = \frac{1}{N}\sum_{i=1}^{N} (P_i - m)(P_i - m)^T$$

If we transform all the points by a matrix A we have:

$$C' = ACA^T$$

Thus A^T must diagonalize C. A is transpose of eigenvectors of C.

What are the principal axis names?

$$R, S, T$$

Construct an OBB.

We could actually transform all the vertices, or dot each vertex with the unit vectors R, S, and T to project them onto the axes and take the min and max values:

$$\langle R, -\min(R \cdot P)\rangle$$

$$\langle S, -\min(S \cdot P)\rangle$$

$$\langle T, -\min(T \cdot P)\rangle$$

$$\langle -R, \max(R \cdot P)\rangle$$

$$\langle -S, \max(S \cdot P)\rangle$$

$$\langle -T, \max(T \cdot P)\rangle$$

Construct a bounding sphere.

$$Q = \frac{P_k + P_l}{2}$$

$$r = \|P_k - Q\|$$

For any point that's outside, expand from tangent point:

$$G = Q - r\frac{P_i - Q}{\|P_i - Q\|}$$

$$Q' = \frac{G + P_i}{2}$$

$$r' = \|P_i - Q'\|$$

Construct a bounding ellipsoid.

Get the axis lengths:

$$a = \max(P_i \cdot R) - \min(P_i \cdot R)$$

$$b = \max(P_i \cdot S) - \min(P_i \cdot S)$$

$$c = \max(P_i \cdot T) - \min(P_i \cdot T)$$

Transform vertex set into cube by moving to RST space, scaling, then moving back to xyz space:

$$M = [R \quad S \quad T] \begin{bmatrix} 1/a & 0 & 0 \\ 0 & 1/b & 0 \\ 0 & 0 & 1/c \end{bmatrix} [R \quad S \quad T]^T$$

Now calculate the bounding sphere: center Q, radius r.

Transform Q back:

$$Q' = M^{-1}Q = [R \quad S \quad T] \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} [R \quad S \quad T]^T$$

Semiaxis lengths are now ar, br, cr.

Construct a bounding cylinder.

Long axis will be R. First get H, the components perpendicular to R:

$$H_i = P_i - (P_i \cdot R)R$$

Then find H with min and max dot product with S, the second-largest axis and thus the one that will inform the radius:

$$Q = \frac{H_k + H_l}{2}$$

$$r = \|H_k - Q\|$$

Check all points and expand:

$$G = Q - r\frac{H_i - Q}{\|H_i - Q\|}$$

$$Q' = \frac{G + H_i}{2}$$

$$r' = \|H_i - Q'\|$$

$$Q_1 = Q + \min(P_i \cdot R)\, R$$

$$Q_2 = Q + \max(P_i \cdot R)\, R$$

Test bounding sphere against view frustum.

Approximate by testing point (center) against all 6 planes to see if distance to any plane is $\leq -r$. If so, object not visible.

Not visible if $Q \cdot L \leq -r$ for any plane

Test bounding ellipsoid against view frustum.

Same, but use effective radius. R, S, T are assumed to have the correct magnitude, *not* unit vectors.

$$r_{eff} = \sqrt{(R \cdot N)^2 + (S \cdot N)^2 + (T \cdot N)^2}$$

Not visible if $Q \cdot L \leq -r_{eff}$ for any plane

Test bounding cylinder against view frustum.

Test *line segment* against all 6 planes.

Unit vector in direction of cylinder long axis:

$$A = \frac{Q_2 - Q_1}{\|Q_2 - Q_1\|}$$

$$r_{eff} = r\sin(\alpha) = r\sqrt{1 - \cos(\alpha)^2} = r\sqrt{1 - (A \cdot N)^2}$$

Calculate $L \cdot Q_1$ and $L \cdot Q_2$. If both are $\leq -r_{eff}$, not visible. If both greater, do nothing. If one greater, one less, calculate new point Q_3 to trim cylinder to such that:

$$L \cdot Q_3 = -r_{eff}$$

$$Q_3(t) = Q_1 + t(Q_2 - Q_1)$$

$$t = \frac{r_{eff} + L \cdot Q_1}{L \cdot (Q_1 - Q_2)}$$

Test bounding box against view frustum.

Choice between point and line. Choose point if the box is close to a cube, or line if one of the axes is much longer since the point approximation can fail to discard the box at the corners.

Point:

Just add up the contributions of each edge:

$$r_{eff} = \frac{1}{2}(|R \cdot N| + |S \cdot N| + |T \cdot N|)$$

Not visible if $Q \cdot L \leq -r_{eff}$ for any plane

Line segment:

$$Q_1 = Q + \frac{1}{2}R$$

$$Q_2 = Q - \frac{1}{2}R$$

$$r_{eff} = \frac{1}{2}(|S \cdot N| + |T \cdot N|)$$

Proceed as with cylinder.

Discuss use of octrees for fast visibility determination.

Calculate r_{eff} for main bounding box B – only need 5 since near and far planes are parallel. Then just halve them for each subdivision level, and no need to check subnodes once a node is marked not visible.

Discuss use of BSP trees for visibility determination.

Arbitrarily oriented planes, unlike octrees. Modern implementations split on whole objects, and align perpendicular to the smallest axis T.

For each splitting plane, we must determine the visibility of each halfspace, and the object stuck to the plane. *Could* test all 8 vertices of view frustum in world space against the plane, but there's a better way: transform the plane into homogeneous clip space and use the cubic symmetry of the view frustum in that space.

$$K' = [(PM)^{-1}]^T K$$

Distance to closest corner of the box given by:

$$d_{max} = |K'_x| + \left|K'_y\right| + |K'_z| + K'_w$$

$$d_{min} = -|K'_x| - \left|K'_y\right| - |K'_z| + K'_w$$

Kareem Omar
kareem_omar@naughtydog.com

Portal systems.

Clip a portal polygon itself to the view frustum. Suppose vertex V_i is inside the frustum, i.e. satisfies:

$$L \cdot V_i > 0$$

And vertex V_{i+1} satisfies:

$$L \cdot V_{i+1} \leq -\epsilon$$

A point W lying on the line segment connecting the two is:

$$W(t) = V_i + t(V_{i+1} - V_i)$$

$$t = \frac{L \cdot V_i}{L \cdot (V_i - V_{i+1})}$$

And discard all vertices on the negative side of the plane.

Reducing the view frustum for testing the region through a portal:

Just planes through 2 verts and the origin:

$$L = \langle \frac{V_{i+1} \times V_i}{\|V_{i+1} \times V_i\|}, 0 \rangle$$

Discard planes from nearby verts:

$$\|V_{i+1} - V_i\|^2 < \epsilon$$

Add planes at corners meeting with small acute angles to prevent lots of false positive visibility tests:

$$A = N_1 + N_2$$

$$B = N_1 \times N_2$$

$$N_3 = \frac{A - (A \cdot B)B}{\|A - (A \cdot B)B\|}$$

Describe parametric vs. geometric continuity.

Parametric: C^n: nth derivatives are equal in both magnitude and direction at the join point.

Geometric: G^n: nth derivatives are nonzero and point in the same direction, but are not necessarily equal in magnitude.

Curvature:

$$\kappa(t) = \left\| \frac{d}{ds}\hat{T}(t) \right\| = \frac{\left\| \frac{d}{dt}\hat{T}(t) \right\|}{\frac{ds}{dt}} = \frac{\|P'(t) \times P''(t)\|}{\|P'(t)\|^3}$$

$$\frac{ds}{dt} = \left\| \frac{d}{dt}P(t) \right\|$$

$$\hat{T}(t) = \frac{T(t)}{\|T(t)\|} = \frac{\frac{d}{dt}P(t)}{\left\| \frac{d}{dt}P(t) \right\|} = \frac{\frac{d}{dt}P(t)}{\frac{ds}{dt}}$$

$$\hat{N}(t) = \frac{\frac{d}{dt}\hat{T}(t)}{\left\| \frac{d}{dt}\hat{T}(t) \right\|}$$

$$\hat{B}(t) = \hat{T}(t) \times \hat{N}(t) \qquad \text{Frenet frame}$$

Collide a sphere with a plane.

Occurs when $L \cdot P = r$, so $L' = \langle N, D - r \rangle$

$$P(t) = P_1 + tV$$

$$L' \cdot P(t) = 0$$

$$L' \cdot P_1 + t(L' \cdot V) = 0$$

$$t = -\frac{L' \cdot P_1}{L' \cdot V}$$

$$C = P(t) - rN$$

Collide a box with a plane.

Same basic idea:

$$L' = \langle N, D - r_{eff} \rangle$$

Vertices of box are:

$$Z = Q(t) \pm \frac{1}{2}R \pm \frac{1}{2}S \pm \frac{1}{2}T$$

Choose signs so that the quantities dotted with N are all negative, so:

$$C = Q(t) - \frac{1}{2}[sgn(R \cdot N)R + sgn(S \cdot N)S + sgn(T \cdot N)T]$$

If a dotted quantity is 0, edge and use $\pm$ in the above. If two dotted quantities are 0, face, and use $\pm$ for both.

General sphere collisions.

See if you hit a face moved outward by r.

If not, see if you hit a cylinder on an edge with radius r.

If not, see if you hit a sphere on a vertex with radius r.

We already know how to do ray-sphere and ray-triangle. Ray-cylinder was done for the aligned case, but these cylinders can be at arbitrary orientations. So:

$$P(t) = S + tV$$

Move the coordinate system so one vertex (E_1) is the origin and the other is E_2. Point P lies on the lateral surface of the infinite cylinder if its distance from the axis $A = E_2 - E_1$ is r.

$$r^2 = P^2 - (proj_A P)^2 = P^2 - \frac{(P \cdot A)^2}{A^2}$$

Replace with ray:

$$S_0 = S - E_1$$

$$a = V^2 - \frac{(V \cdot A)^2}{A^2}$$

$$b = S_0 \cdot V - \frac{(S_0 \cdot A)(V \cdot A)}{A^2}$$

$$c = S_0^2 - r^2 - \frac{(S_0 \cdot A)^2}{A^2}$$

The length of the projection of the collision point relative to E_1 onto the axis is:

$$L = \frac{[P(t) - E_1] \cdot A}{\|A\|}$$

Which must be between 0 and $\|A\|$, i.e.:

$$0 < [P(t) - E_1] \cdot A < A^2$$

Do bounding sphere test (R+r) first!

Collision of two spheres.

$$A = P_1 - Q_1$$

$$B = V_P - V_Q$$

$$d^2 = A^2 + 2t(A \cdot B) + t^2 B^2$$

Exact collision:

$$t = \frac{-(A \cdot B) - \sqrt{(A \cdot B)^2 - B^2\left[A^2 - \left(r_P + r_Q\right)^2\right]}}{B^2}$$

Fast check:

$$d^2 = A^2 - \frac{(A \cdot B)^2}{B^2}$$

Partitioning sine and cosine:

$$\sin(\alpha + \beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$