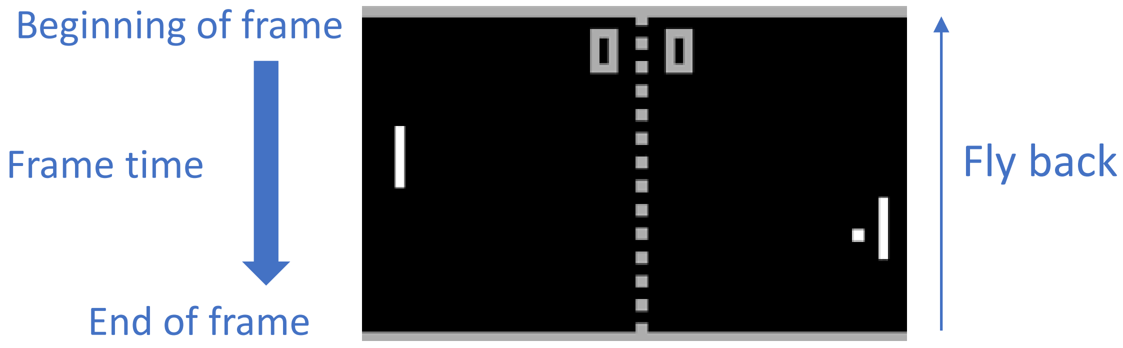
Week 8: Profiling & Optimisation

COMP270: Mathematics for 3D Worlds & Simulations

BSc(Hons) Computing for Games

Time is of the essence

A brief history of (frame) time



- 60fps means 16.67ms per frame…
  - Origin: US & Japanese TV systems run NTSC
    - Gives 30fps
    - Updates at 60fps
    - Electrical power is generated at 60Hz
  - Early games drew using the CRT scan directly
    - "Racing the Beam: The Atari Video Computer System", Ian Bogost and Nick Montfort, MIT Press
    - Drawing took up all the time during the scan; other work had to be done during the "fly back".

We still follow this pattern of separating the core/game logic from the drawing

  - PAL is 50fps; has a higher vertical resolution
    - But satisfy the most demanding!
    - NB can't run NTSC code directly on PAL (e.g. runs at $^5/_6$ speed): need to *port*
    - VR needs even faster frame-rate – 100fps for two screens – as do hi-res displays
  - More on the history here: https://www.howtogeek.com/428987/whats-the-difference-between-ntsc-and-pal/

Signs that something isn’t quite right

- The game “feels slow”
  - Doesn’t meet required frame rate
- The fan is always running/machine gets hot
- It takes a long time to load a level
- It takes a long time to build the code

Types of optimisation

- Space vs. time
  - Often improve one at some cost to the other
- CPU, memory and GPU
- Micro vs. macro
  - Looking for “hot spots” and focussing on a few lines at a time
    - Pareto Principle (80:20 rule): most of the problems will come from a small amount of code
  - Looking at the overall design/structure for systemic issues
    - Design anti-patterns – e.g. “God objects”/”Swiss Army classes”, long functions
    - Design pattern misuse/overuse
    - Over-engineered solutions

UML, flow charts

Maths is complex

Breaking things down

Taking the dot product of two *n*-dimensional vectors involves:

- *n* multiplications
- *n*-1 additions

For 3D vectors:
3 multiplications and
2 additions

Multiplying two *nxn* matrices (= *nxn* *n*-dimensional dot products) involves:

- n^3 multiplications
- $n^3 - n^2$ additions

For general 4x4 matrices:
64 multiplications and
48 additions

For homogeneous 4x4 matrices:
36 multiplications and
27 additions

Breaking things down

Finding the length of an *n*-dimensional vector involves:

- *n* multiplications
- *n* - 1 additions
- One square root

For 3D vectors:
3 multiplications, 2 additions
– and a square root

Function	Time complexity (best case*)
multiplication	$O(n^{1.585})$
sqrt	$O(M(n))$
trig, exp/log	$O(M(n)\log n)$

* For non-large input
Where $M(n)$ is the time complexity of the chosen multiplication algorithm
Source: https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations

Normalising an *n*-dimensional vector involves:

- All the above, plus
- *n* divisions!

Breaking things down

Finding the angle between two vectors, $\theta = \text{acos}\left(\frac{\boldsymbol{a}\cdot\boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|}\right)$ involves:

- One dot product between two vectors
- Finding the lengths of two vectors
- One division
- One *acos*

For 3D vectors:
9 multiplications, 1 division,
6 additions, 2 square roots –
and an *acos*…

Taking the cross product of two 3D vectors involves:

- 6 multiplications
- 3 subtractions

Breaking things down

Multiplying two quaternions $\boldsymbol{q}_1\boldsymbol{q}_2 = [w_1w_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \quad w_1\boldsymbol{v}_2 + w_2\boldsymbol{v}_1 + \boldsymbol{v}_1 \times \boldsymbol{v}_2]$ involves:

- 7 scalar/component multiplications
- 1 3D dot product = 3 multiplications and 2 additions
- 1 3D cross product = 6 multiplications and 3 subtractions
- 6 component-wise additions and 1 subtraction

Total:
16 multiplications,
8 additions and
4 subtractions

x2 to
apply to
vector!

Quaternion SLERP (algebraic form): $slerp(\boldsymbol{q}_0, \boldsymbol{q}_1, \mathrm{t}) = (\boldsymbol{q}_0\boldsymbol{q}_1{}^{-1})^t\boldsymbol{q}_0$

- Two quaternion products
- One quaternion exponent, $\boldsymbol{q}^t = \exp(t\log\boldsymbol{q}) = \exp(t\,[0 \quad \alpha\hat{\boldsymbol{n}}])$
  $= \exp([0 \quad \alpha t\hat{\boldsymbol{n}}]) = [\cos\alpha \quad \hat{\boldsymbol{n}}\sin t\alpha]$

SLERP derivation

Standard linear interpolation (LERP):

$$\Delta a = a_1 - a_0$$
$$lerp(a_0, a_1, t) = a_0 + t\Delta a$$

1. Compute the difference between the values
2. Take a fraction of the difference
3. Adjust the original value by the fraction of the difference

SLERP derivation (cont.)

Analogous steps for interpolating between orientations:

1. *Compute the difference between the values:*
   The angular displacement from $\boldsymbol{q}_0$ to $\boldsymbol{q}_1$ is given by the *quaternion difference*,
   $$\Delta\boldsymbol{q} = \boldsymbol{q}_1 {\boldsymbol{q}_0}^{-1}$$
2. *Take a fraction of the difference:*
   Given by quaternion exponentiation, $(\Delta\boldsymbol{q})^t$
3. *Adjust the original value by the fraction of the difference:*
   Combine the rotations $\boldsymbol{q}_0$ and $(\Delta\boldsymbol{q})^t$ via quaternion multiplication,
   $$(\Delta\boldsymbol{q})^t \boldsymbol{q}_0$$

Complex things take time

Adding things up

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
comp270-worksheet-C.exe (PID: 15224)	17671 (100.00%)	0 (0.00%)
Vector3D::dot	187 (1.06%)	187 (1.06%)
Camera::getClosestIntersectedObject	1253 (7.09%)	155 (0.88%)
Plane::getIntersection	599 (3.39%)	151 (0.85%)
Sphere::getIntersection	260 (1.47%)	117 (0.66%)
fabsf	100 (0.57%)	100 (0.57%)
Point3D::operator-	147 (0.83%)	86 (0.49%)
Vector3D::Vector3D	84 (0.48%)	84 (0.48%)
std::_Vector_alloc<std::_Vec_base_types<Colour,s...	134 (0.76%)	77 (0.44%)
std::_Compressed_pair<std::allocator<Colour>,st...	67 (0.38%)	67 (0.38%)
std::_Vector_alloc<std::_Vec_base_types<Object *,...	166 (0.94%)	58 (0.33%)
Point3D::Point3D	47 (0.27%)	47 (0.27%)
std::vector<Object *,std::allocator<Object *> >::o...	110 (0.62%)	40 (0.23%)
std::vector<Object *,std::allocator<Object *> >::si...	128 (0.72%)	37 (0.21%)
Vector3D::operator*	52 (0.29%)	29 (0.16%)
fabs	124 (0.70%)	24 (0.14%)
Point3D::operator+	71 (0.40%)	24 (0.14%)
Application::render	2102 (11.90%)	16 (0.09%)
std::_Vector_alloc<std::_Vec_base_types<Vector3...	42 (0.24%)	16 (0.09%)
std::_Default_allocator_traits<std::allocator<Colou...	23 (0.13%)	16 (0.09%)
Image::getPixel	30 (0.17%)	15 (0.08%)
std::vector<Vector3D,std::allocator<Vector3D> >:...	30 (0.17%)	15 (0.08%)

Camera::getClosestIntersectedObject

comp270-worksheet-C.exe

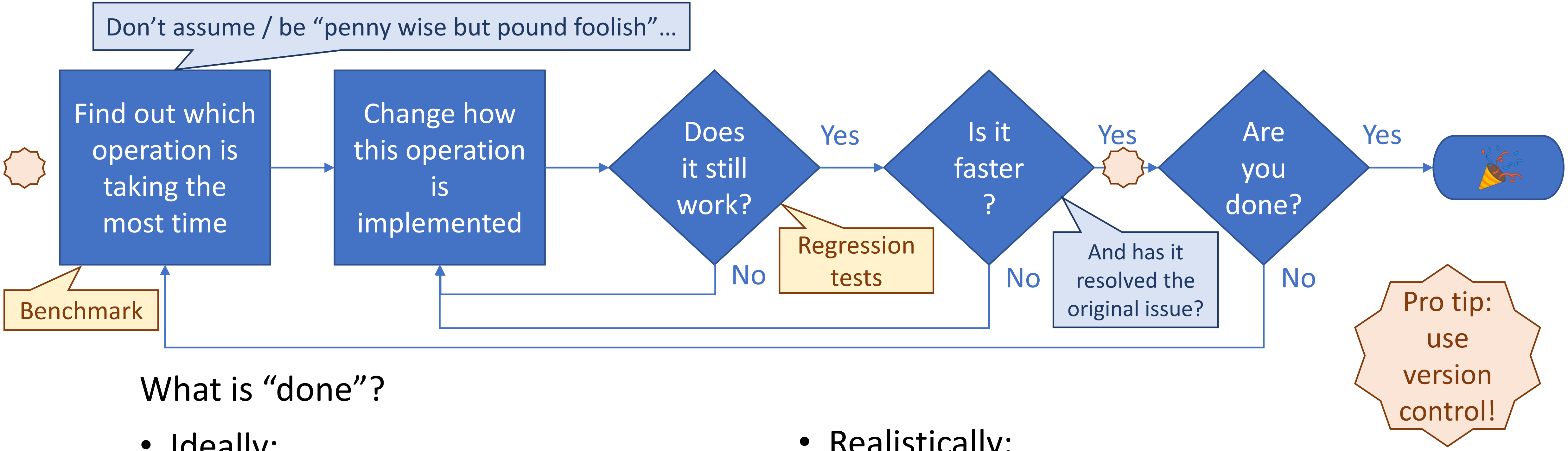
Calling Functions		Current Function		Called Functions	
Camera::updateScreenBuffer	1253 (7.09%)	Camera::getClosestIntersectedObject	1253 (7.09%)	Plane::getIntersection	599 (3.39%)
		Function Body	155 (0.88%)	Sphere::getIntersection	260 (1.47%)

c:\users\kb242181\documents (local)\visual studio\comp270-worksheet-c\comp270-worksheet-c\camera.cpp:158

```
              153  // Params:
              154  //   raySrc  starting point of the ray (input)
              155  //   rayDir  direction of the ray (input)
              156  //   objects list of pointers to objects to test (input)
              157  const Object* Camera::getClosestIntersectedObject(const Point3D& raySrc, const Vector3D& rayDir, const std::vector<Object*>& objects) co
              158  {
   4 (0.02%)  159      float distToNearestObject = FLT_MAX;
              160      const Object* nearestObject = nullptr;
 149 (0.84%)  161      for (unsigned objIdx = 0; objIdx < objects.size(); ++objIdx)
              162      {
              163          float distToFirstIntersection = FLT_MAX;
              164          if (objects[objIdx]->getIntersection(raySrc, rayDir, distToFirstIntersection)
1089 (6.16%)  165              && distToFirstIntersection < distToNearestObject)
              166          {
   1 (0.01%)  167              nearestObject = objects[objIdx];
   8 (0.05%)  168              distToNearestObject = distToFirstIntersection;
              169          }
              170      }
              171
   1 (0.01%)  172      return nearestObject;
   1 (0.01%)  173  }
```

Avoidance strategies

General approach

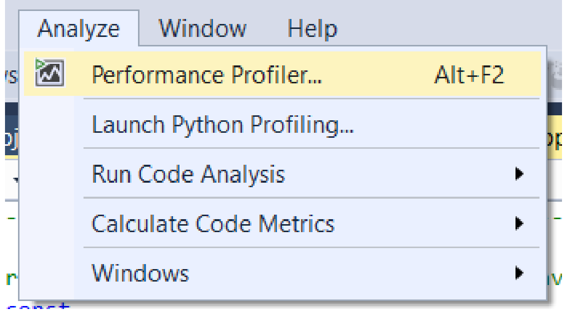


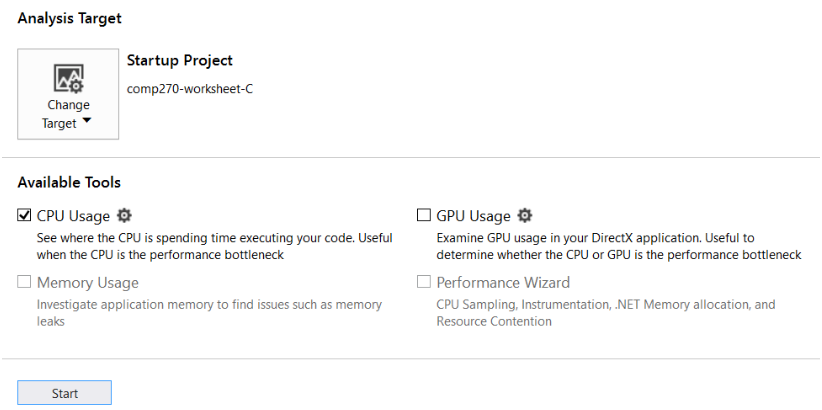
What is "done"?

- Ideally:
    - Each operation is implemented as efficiently as possible
    - Only essential operations are carried out
- Realistically:
    - The code runs fast enough to support the desired frame rate
    - (You've run out of time/ideas for how to improve the efficiency…)

Step 1: Finding the time

- Use built-in profilers, e.g. Visual Studio:
  - https://docs.microsoft.com/en-us/visualstudio/profiling/
- Alternatively, profiling packages are available, e.g. Shiny:
  - https://sourceforge.net/projects/shinyprofiler/
- Use quantitative data:
  - Frame rate/time
  - Memory use
  - Loading time
  - Counter for the number of times a function is called
- Create a reusable test case
- Keep records of previous results to refer back to!





Benchmark

- A point of reference for the game, which serves as a standard for comparison
- A good benchmark should be:
  - Consistent between runs
  - Quick to carry out
  - Representative of an actual game situation
  - Responsive to changes

Step 2: Reducing the cost

1. Do less:
    - Get rid of pointless code (especially in older projects)
    - Precompute/cache results
        - e.g. m_pixelRays in Worksheet C
    - Early exits with inexpensive tests
        - e.g. using the dot product to rule out invisible/non-intersecting objects; performing tests on simpler "bounding shapes/volumes" first
    - Avoid creating new objects
        - … but make sure you delete the old ones!

Increases storage cost; requires memory look-ups

May increase storage cost/memory look-ups

Step 2: Reducing the cost

2. Do it more efficiently:
    - Pre-process data and/or use a more appropriate data structure
        - e.g. sorting/using a set rather than a vector
        - Spatial data structures: octree, kd-tree, bounding volume hierarchy (BVH)
    - Try a different algorithm/reformulate the computation
        - To reduce the number of calculations
            - e.g. SLERP, ray-sphere intersection test
        - To increase memory coherence/minimise load times

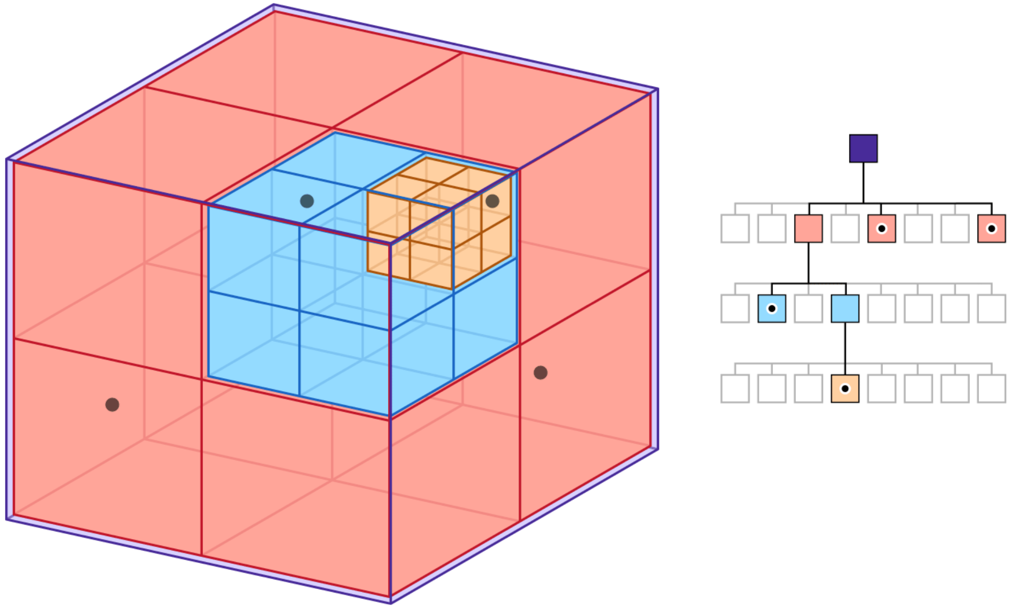
Choosing an appropriate data structure

Memory model	Examples	Use when	Look-up/search time	Insertion time
Sequential	Array, vector	Data is static: few insertions/deletions	O(1) for known look-up O(n) for searching	O(n)
Linked	List, linked list, DAG	Data is dynamic: many insertions/deletions	O(n)	O(1) (unless sorted)
Associative	Dictionary, map, binary search tree	Data is not indexed by integer (and ideally static)	O(1) to O(log$n$)	O(1) to O(log$n$)

More here: https://www.bigocheatsheet.com/

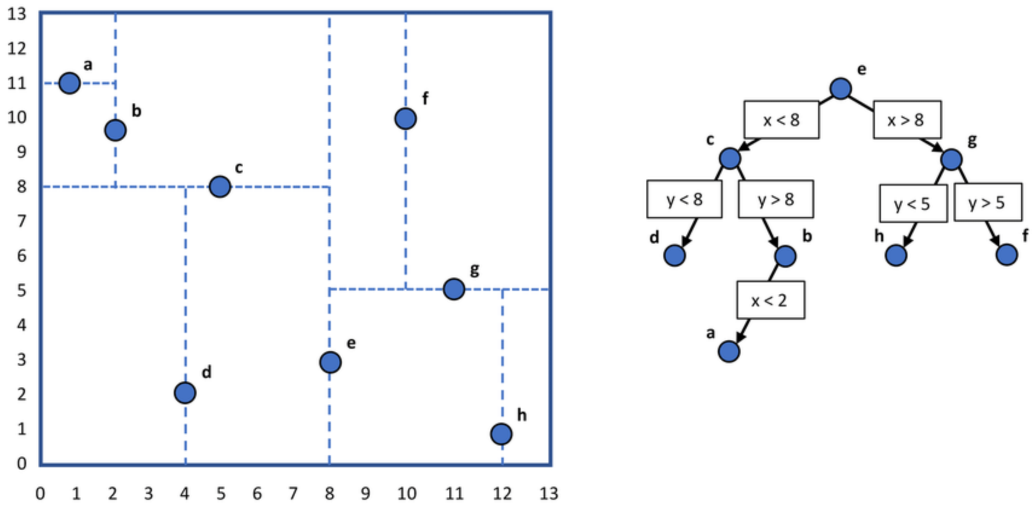
Some spatial data structures

- *Octree*: a tree data structure in which each internal node has exactly eight children.
  - Recursively subdivide 3D space into octants (2D space equivalent: *quadtree*)
  - Store objects in the "cell" corresponding to their position in space (leaf nodes are points)
  - https://www.gamedev.net/articles/programming/general-and-gameplay-programming/introduction-to-octrees-r3529/



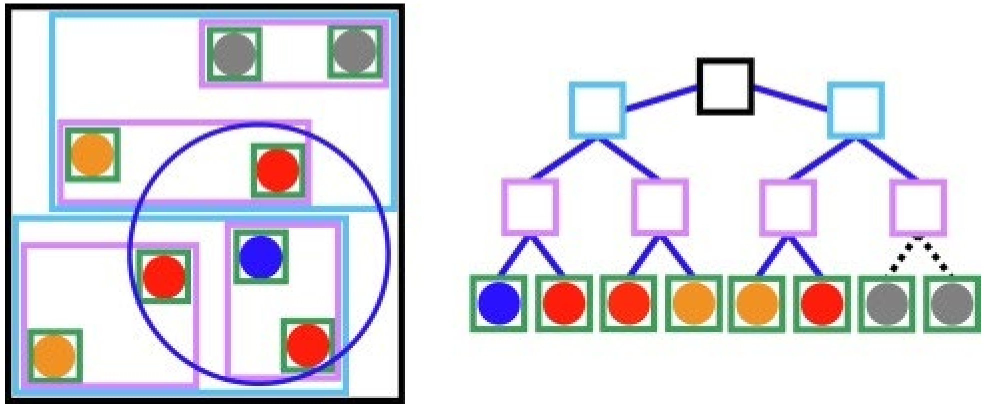
Some spatial data structures

- K-*d tree*: a binary space-partitioning data structure for organising points in a *k*-dimensional space.
    - Every leaf node is a *k*-dimensional point
    - Every non-leaf node represents a division of space into half-spaces by a hyperplane
    - Points to the left of the hyperplane are stored in the left subtree, etc.
    - https://www.geeksforgeeks.org/k-dimensional-tree/



Some spatial data structures

- *Bounding volume hierarchy (BVH):* a tree structure containing a set of geometric objects, which are enclosed in *bounding volumes* that are the leaf nodes.
  - Partitions the objects instead of space
  - Used to accelerate collision detection, ray intersection etc.
  - Groups of nearby bounding volumes are enclosed in larger bounding volumes
  - A single complex object may have several bounding volumes for separate components
  - Some cost to maintaining/updating as objects move around!



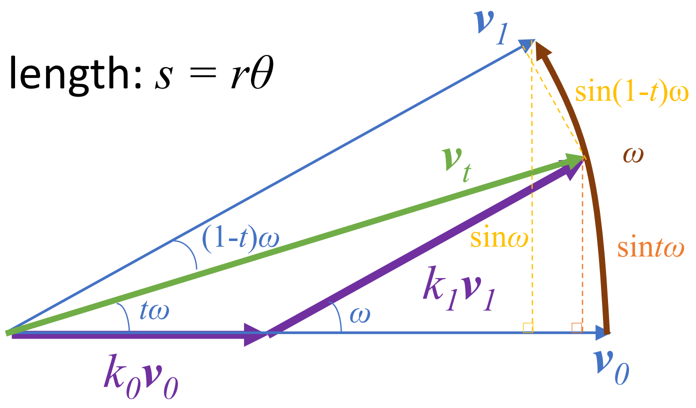
- http://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies.html

Reformulation example: SLERP

Interpret quaternions as existing in a 4D Euclidean space:

- Since all rotation quaternions are unit length, they "live" on the surface of a 4D hypersphere
  - Interpolate around the arc along the surface of the hypersphere, which connects the quaternions:

Arc length: $s = r\theta$



- Express $\boldsymbol{v}_t$ as a linear combination of $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$,

$$\boldsymbol{v}_t = k_0 \boldsymbol{v}_0 + k_1 \boldsymbol{v}_1$$

- Use geometry to find values of k_0 and k_1

$$\sin\omega = \frac{\sin t\omega}{k_1} \Rightarrow k_1 = \frac{\sin t\omega}{\sin\omega}$$

$$k_0 = \frac{\sin(1-t)\omega}{\sin\omega}$$

Reformulation example: SLERP (cont.)

$$\boldsymbol{v}_t = \frac{\sin(1-t)\omega}{\sin\omega}\boldsymbol{v}_0 + \frac{\sin t\omega}{\sin\omega}\boldsymbol{v}_1$$

Extended into quaternion space: $\mathrm{slerp}(\boldsymbol{q}_0, \boldsymbol{q}_1, t) = \frac{\sin(1-t)\omega}{\sin\omega}\boldsymbol{q}_0 + \frac{\sin t\omega}{\sin\omega}\boldsymbol{q}_1$

How to find ω?

- Quaternion dot product, $\boldsymbol{q}_1 \cdot \boldsymbol{q}_2 = [w_1 \quad \boldsymbol{v}_1] \cdot [w_2 \quad \boldsymbol{v}_2] = w_1 w_2 + \boldsymbol{v}_1 \cdot \boldsymbol{v}_2$
  $= [w_1 \quad (x_1 \quad y_1 \quad z_1)][w_2 \quad (x_2 \quad y_2 \quad z_2)] = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2$
  $=$ the w component of the quaternion difference $\boldsymbol{q}_2 \boldsymbol{q}_1{}^*$
  $= \cos\left(\frac{\theta}{2}\right)$

SLERPing problems

1. The two quaternions $\boldsymbol{q}$ and $-\boldsymbol{q}$ represent the same orientation, but may give different results when SLERPed (because a 4D hypersphere has a different topology from Euclidean space)
   - Solution: choose signs of $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ so that the dot product is non-negative
     = selecting the shortest rotational arc between them.
2. If $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ are very close, then ω is very small and so is $\sin\omega$, which can cause problems with the division.
   - Use simple linear interpolation in these cases.

Reformulation example: ray-sphere intersection

From 2 weeks ago – solve the following equation for s:

$$\left(v_x{}^2 + v_y{}^2 + v_z{}^2\right)s^2 + 2\left(a_x v_x + a_y v_y + a_z v_z\right)s + \left(a_x{}^2 + a_y{}^2 + a_z{}^2 - r^2\right) = 0$$

to give the distance of the intersection point along the line $\boldsymbol{x} = \boldsymbol{a} + s\boldsymbol{v}$

Reinterpret the problem geometrically:

$s_0 = t - t_c$

$s_1 = t + t_c$

$d = \sqrt{\|\boldsymbol{l}\|^2 - t^2}$

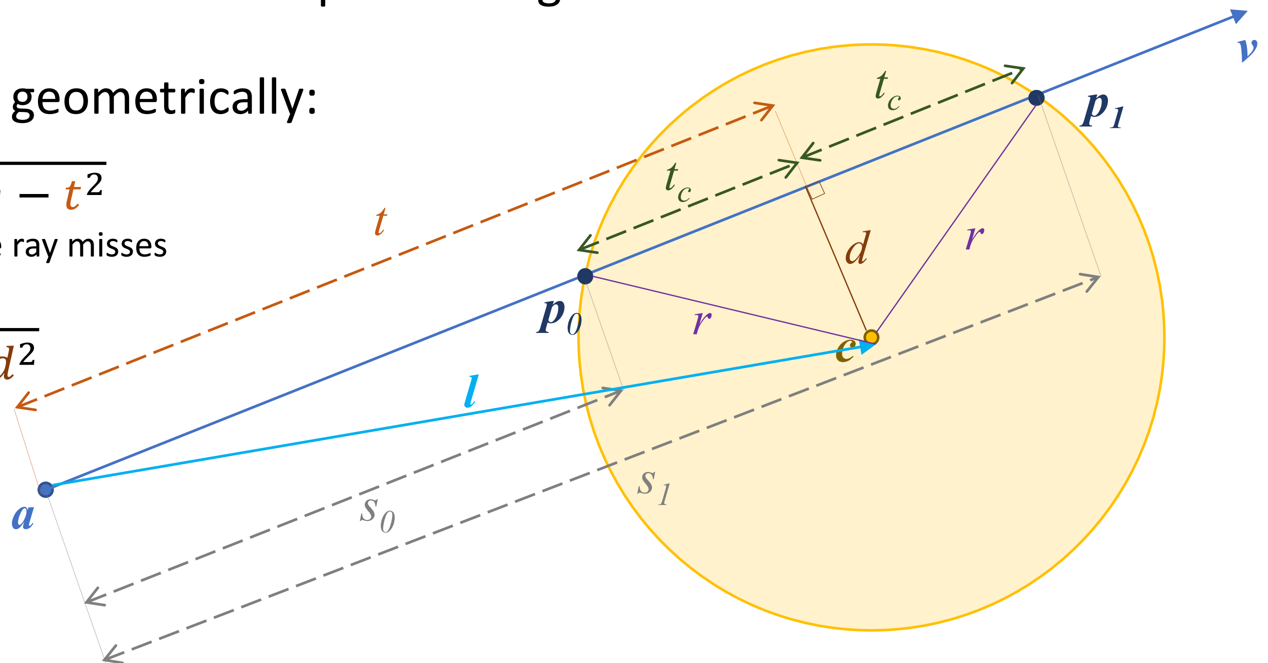
NB $d > r$ means the ray misses the sphere

$\boldsymbol{l} = \boldsymbol{c} - \boldsymbol{a}$

$t = \boldsymbol{l} \cdot \boldsymbol{v}$ (projection)

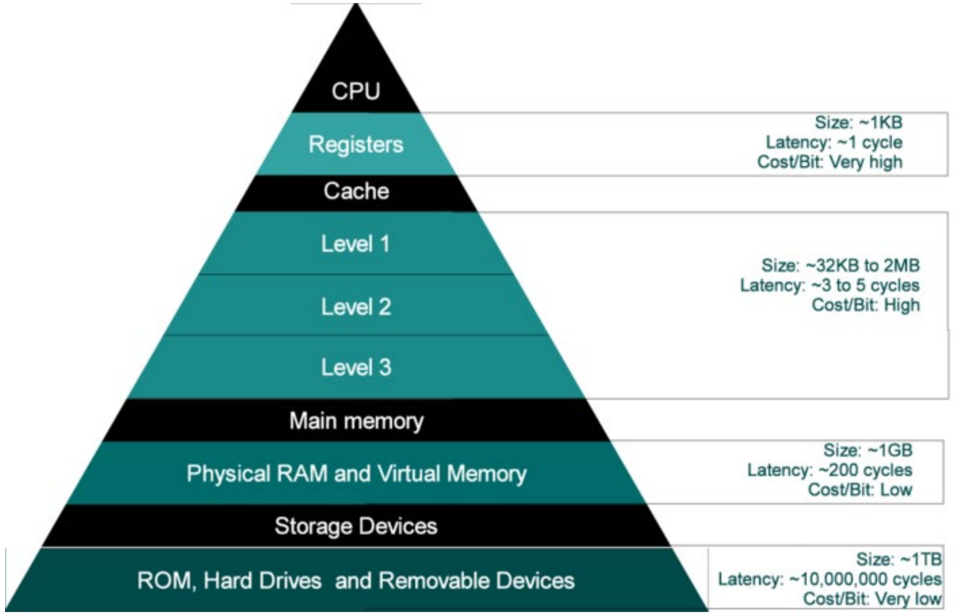
NB $t < 0$ means the intersection is "behind" $\boldsymbol{a}$

$t_c = \sqrt{r^2 - d^2}$



Memory optimisation

- Problems with speed may be due to poor memory management
- Different techniques for increasing efficiency, based on memory structure:



Registers and cache

- Registers:
  - The only memory directly on the CPU
  - “Scratch pad” for current calculations
  - Slow to transfer data from RAM (typically 26 cycles + 57ns)
- Cache:
  - Intermediate between CPU and all off-board memory
  - Usually different levels, with different capabilities depending on the CPU
    - e.g. Intel i-7 4770 (Haswell) has:
      - L1 data cache: 32KB with 4 cycles for simple access
      - L1 instruction cache: 32Kb
      - L2 cache: 256KB with 12 cycles
      - L3 cache: 8MB with 26 cycles

Improving memory management

- Reduce footprint
  - Use smaller data types, e.g. bit flags to replace several booleans
  - Can fit more in the cache at once!
- Use memory-aligned data
  - Read faster by the CPU
  - An *n*-byte piece of data is memory aligned if its starting address is evenly divisible by *n*
    - e.g. a 32-bit int is aligned if its starting address is 0x04000000, but not 0x04000002
  - Order members of class/struct from largest to smallest
- Structure code so as to reduce memory traversal
- Increase cache hits/avoid misses

Compiler and linker rules

- Assumptions you can safely make to avoid cache misses:
  - The machine code for a single function is contiguous in memory
  - Functions are laid out in memory in the order they appear in the cpp file
  - Functions in the cpp file are always contiguous
- So:
  - Keep high performance code as small as possible
  - Avoid calling functions from a performance critical section of code
  - If you do have to call a function, place it as close as possible (never in another translation unit/source file)
  - Use inline functions – judiciously!
    - Overuse can lead to code bloat and increase cache misses

Tips and tricks

- Make use of version control – commit each time you get it working!
- Do most of your profiling in release build
  - though debug can be helpful to start with
- Before starting to optimise, make sure that your code:
  1. Works fully – later changes/features may invalidate your efforts!
  2. Is as clean as possible:
     - No unused variables, functions etc.
     - Every operation has a purpose
     - No memory leaks, crashes etc.
     - No “smells”: https://en.wikipedia.org/wiki/Code_smell

Premature optimisation

- Focus on one section/problem at a time, creating a specific and consistent test case
  - Possibly hard-coding/limiting other aspects to narrow scope/exaggerate the situation
  - Make sure to revert any code changes afterwards!

Worksheet C: submit by Monday 25th if you'd like feedback!