Light probe interpolation using tetrahedral tessellations

Robert Cupisz
@robertcupisz

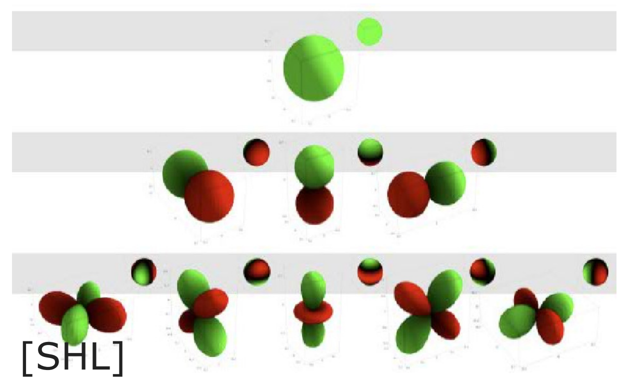
Graphics Programmer
Unity Technologies

GAME DEVELOPERS CONFERENCE® 2012
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9

The purpose of this talk is to present an alternative light probe interpolation method. I'll start with discussing the problems of currently used solutions and proceed to explaining our technique.

Light probes – recap

- Samples of the offline GI
- Spherical Harmonics encode the directionality of the irradiance
- Reconstructing SH:

$\sum$  [SHL] $*$

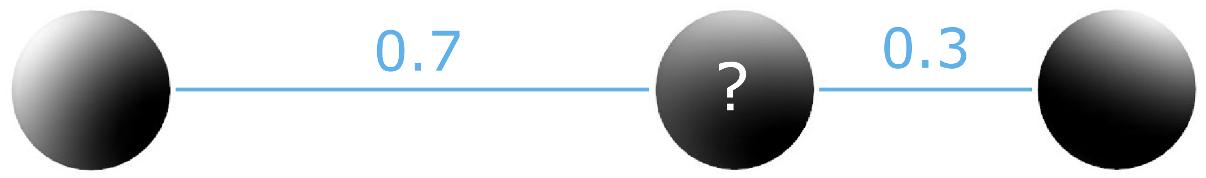
1.2

-0.9 -0.3 1.2

-0.2 0.4 -1.2 -0.4 -0.2

$=$

- SH interpolation:



1.2

-0.9 -0.3 1.2

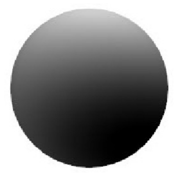
-0.2 0.4 -1.2 -0.4 -0.2

$*$ 0.7 $+$

0.8

-1.3 -0.2 -0.5

1.0 0.5 -0.8 0.2 -1.1

$*$ 0.3 $=$ 

First a small recap.

Light probes are usually samples of the Global Illumination calculated by the lightmapper. Irradiance is sampled at a number of points in the scene and encoded somehow for each of those points. Spherical Harmonics are typically used for encoding, as they can nicely capture the low frequency directionality of irradiance.

An SH probe is stored as coefficients for the spherical harmonics basis functions. In the image you can see the 9 basis functions and their corresponding coefficients. Reconstructing an SH probe is just calculating a linear combination of the functions and the coefficients, which gives a function that can be evaluated for any direction and returns the light intensity for that direction.

Since a probe is reconstructed by a linear combination of coefficients and basis functions, interpolating two probes can be done just by interpolating their coefficients.

Light probe interpolation – recap

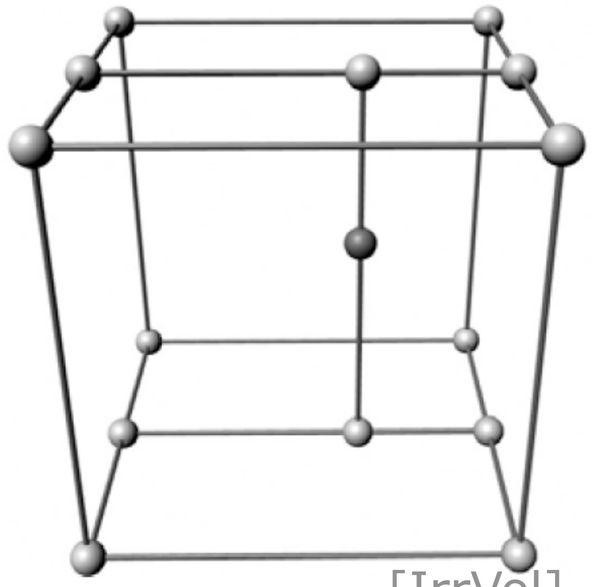
- A dynamic object needs an interpolated probe at it’s position
  - **Which probes** and with **what weights** to take?

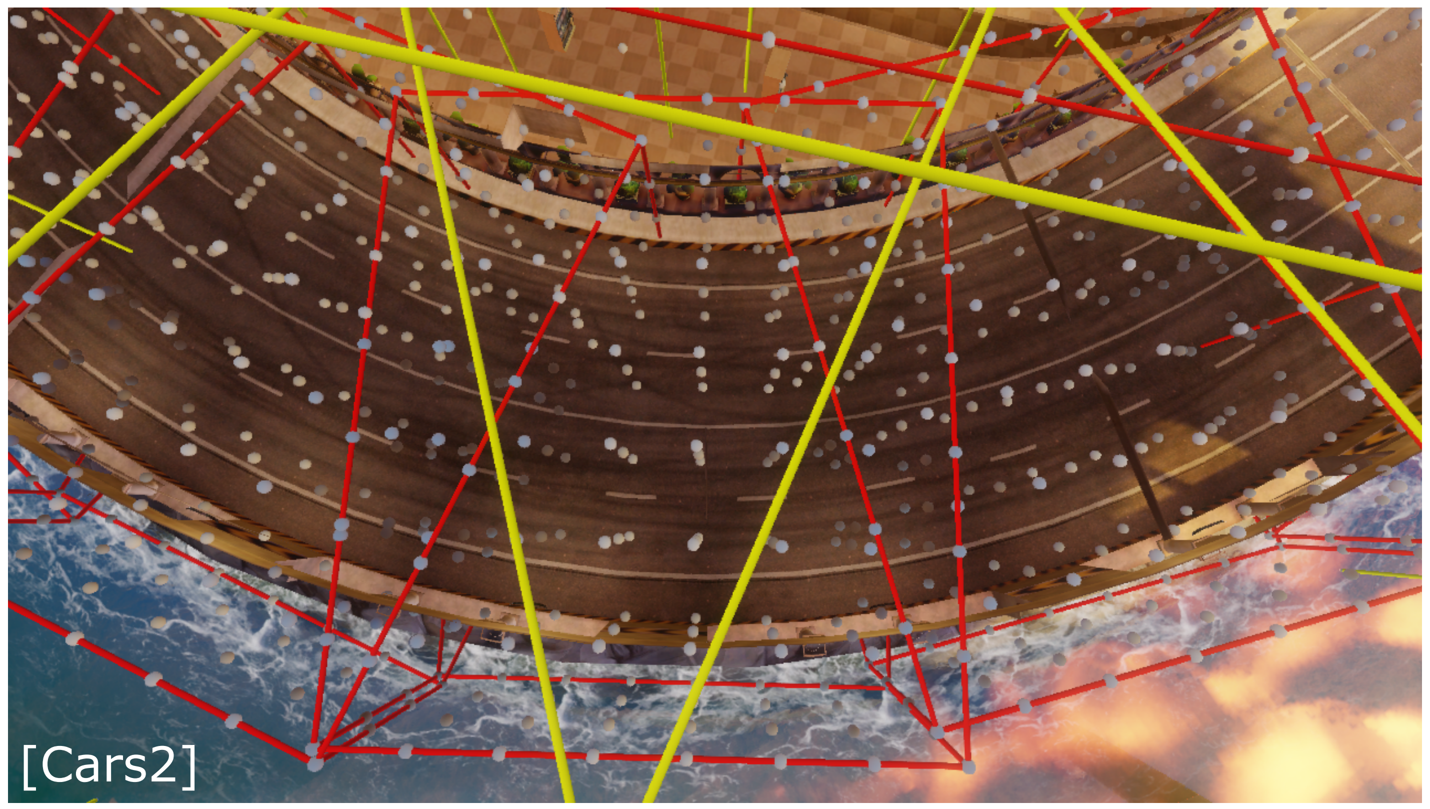
To light a dynamic object we need to find an interpolated probe at it’s center. The problem we need to solve is:

Of all the probes we have calculated for the scene, which probes and with what weights should we take to calculate that interpolated probe?

Light probe interpolation – recap

- Standard approach: place probes so that trilinear interpolation is easy
  - Uniform grid everywhere
  - OBBs filled with uniform grids
    - (+some magic between OBBs)


[IrrVol]

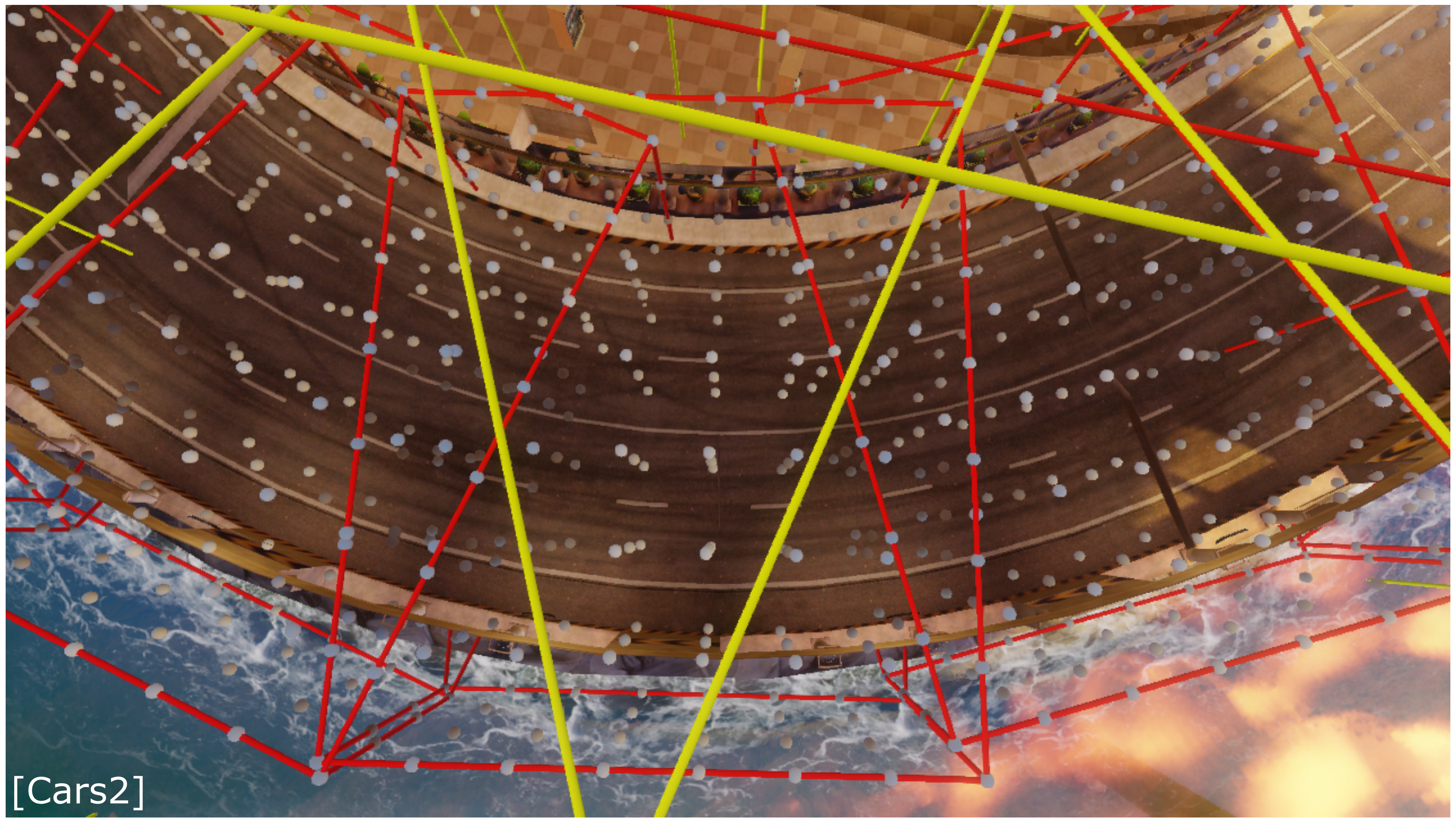

[Cars2]

[Cars2]

Typically probes are arranged in 3D regular grids to enable trilinear interpolation.
To avoid filling the entire space with uniform density of probes, some schemes apply adaptive subdivision in octree-like structures, but that complicates interpolation.
The standard approach seems to be a number of OBBs filled with regular grids. Since the OBBs can be placed anywhere, we're almost back to the original issue of having to interpolate between n different points in space whenever our object's position falls between the OBBs or in an area where the OBBs overlap.

Light probe interpolation – recap

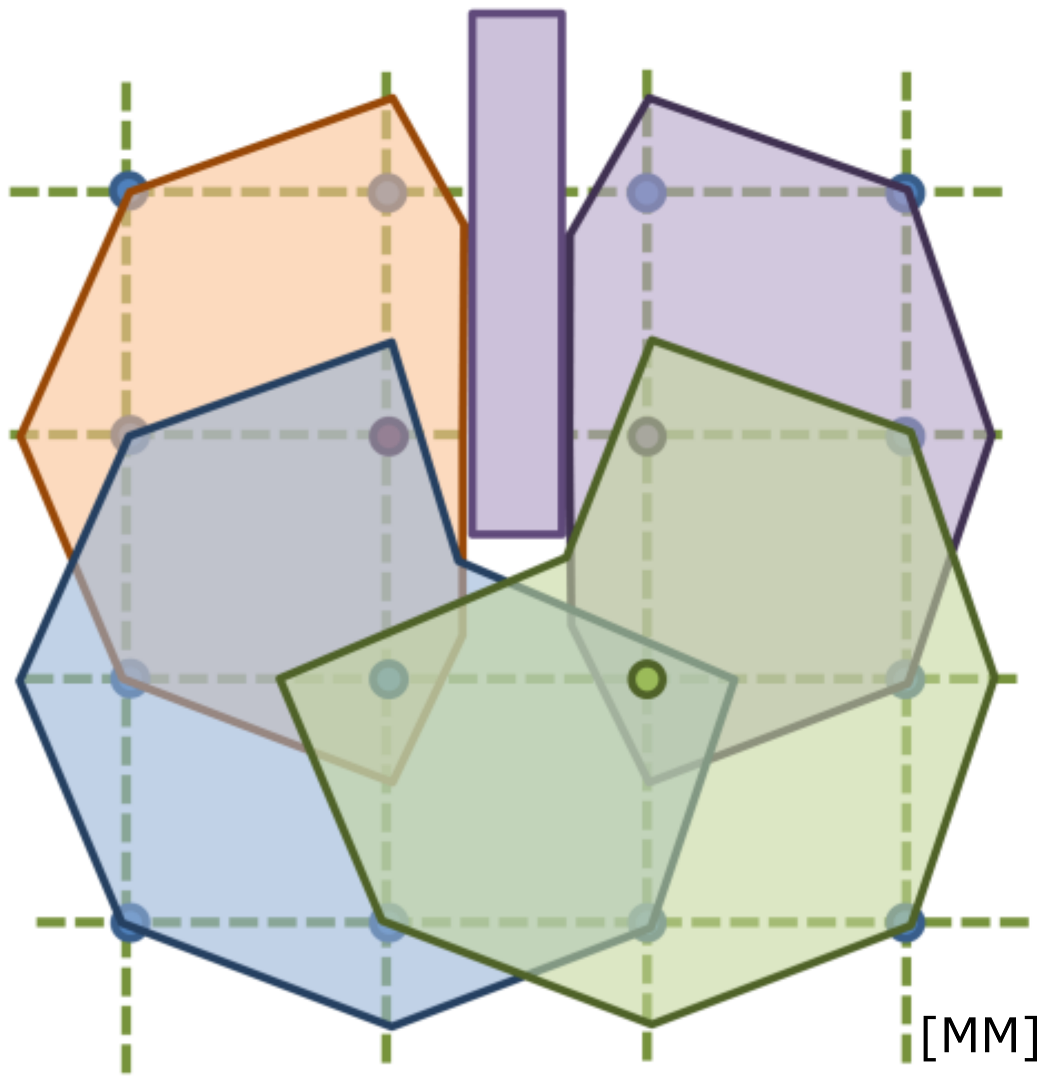
- Problem 1: So. Much. Data.
  - Undersampling AND oversampling

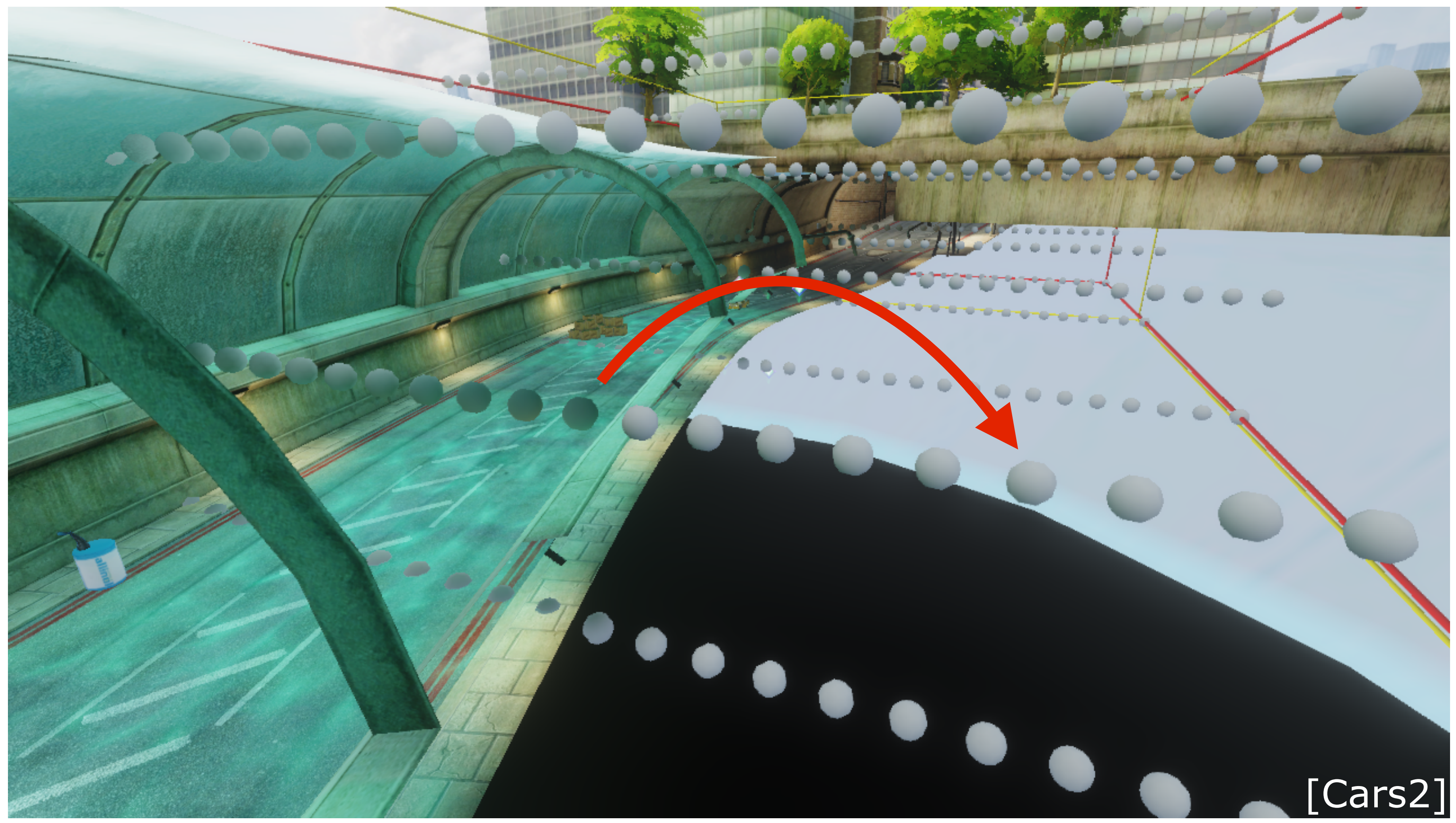


The first issue with regular grids is that they don't account for the fact, that some areas require high-density sampling, while in other a single probe would be sufficient. The grid density in the end becomes something in between, undersampling the interesting areas and oversampling elsewhere, which is a waste.
Placing OBBs with different densities doesn't solve the issue, as the granularity of variations is usually much finer.

Light probe interpolation – recap

- Problem 2: Visibility.
  - Milo and Kate:
  - Cars 2:





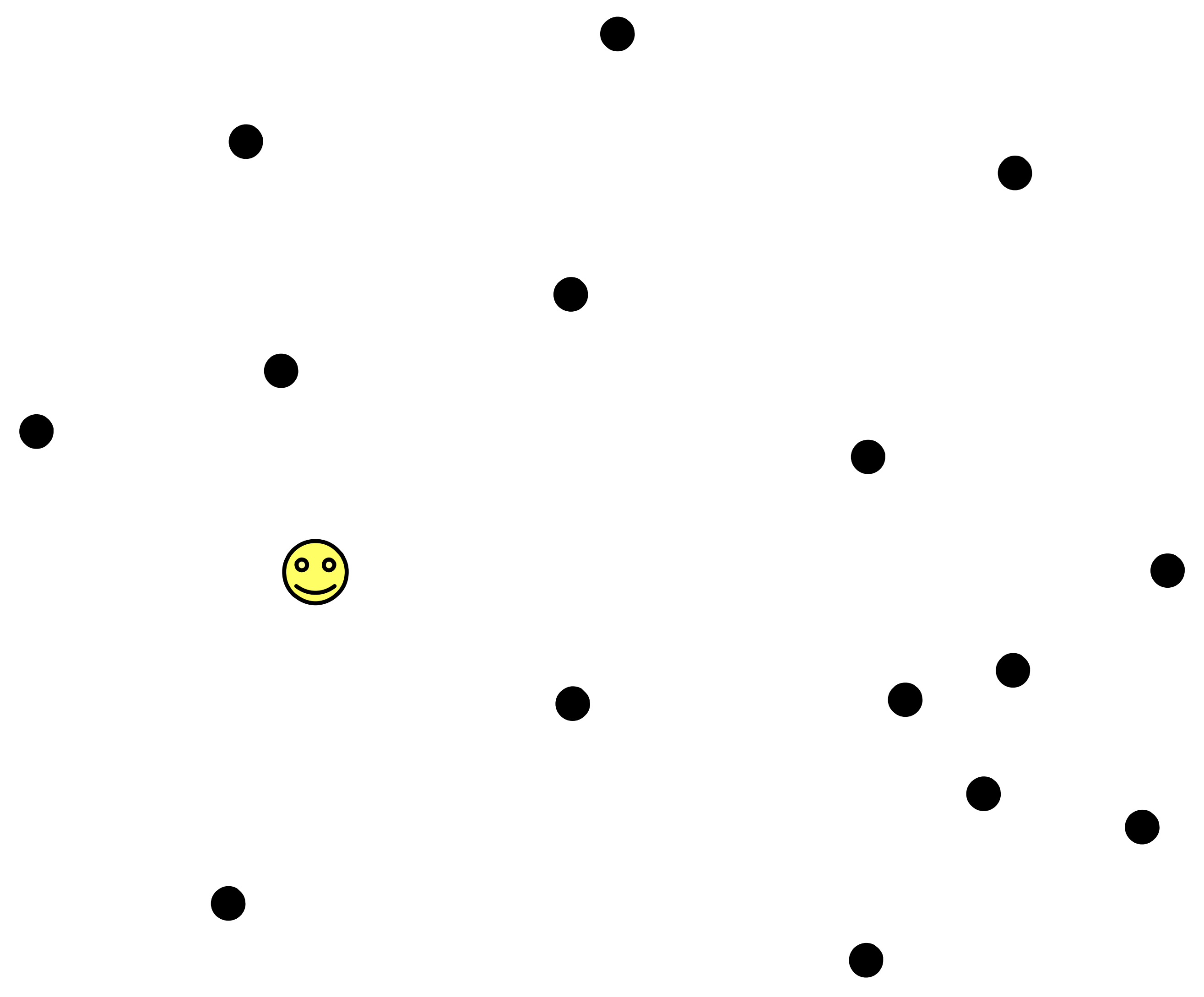
When probes are arranged in regular grids, that inevitably puts some of them within obstacles.
The result is characters becoming dark as they approach walls with black probes baked inside.
A related issue is that potentially completely differently coloured probes from a different lighting environment on the other side of the wall will start interpolating in when the character stands too close to the wall.
Milo and Kate solves it by baking per-probe explicit visibility information, which limits each probe's influence up to the nearest obstacle. This seems to be what most games do nowadays.

Cars 2 uses a simple hack: since the cars always stay on the track side of any wall, any probes outside of the track can be overwritten with the outmost probe that's still on the track. Even if they get interpolated in, they still have the same value.

Some engines sample n nearest probes, raycast to test visibility and fade over time to avoid popping. This is wrong on so many levels, we don't have time to discuss it now ;)

Alternative solution

- In 2D: which probes with what weights?

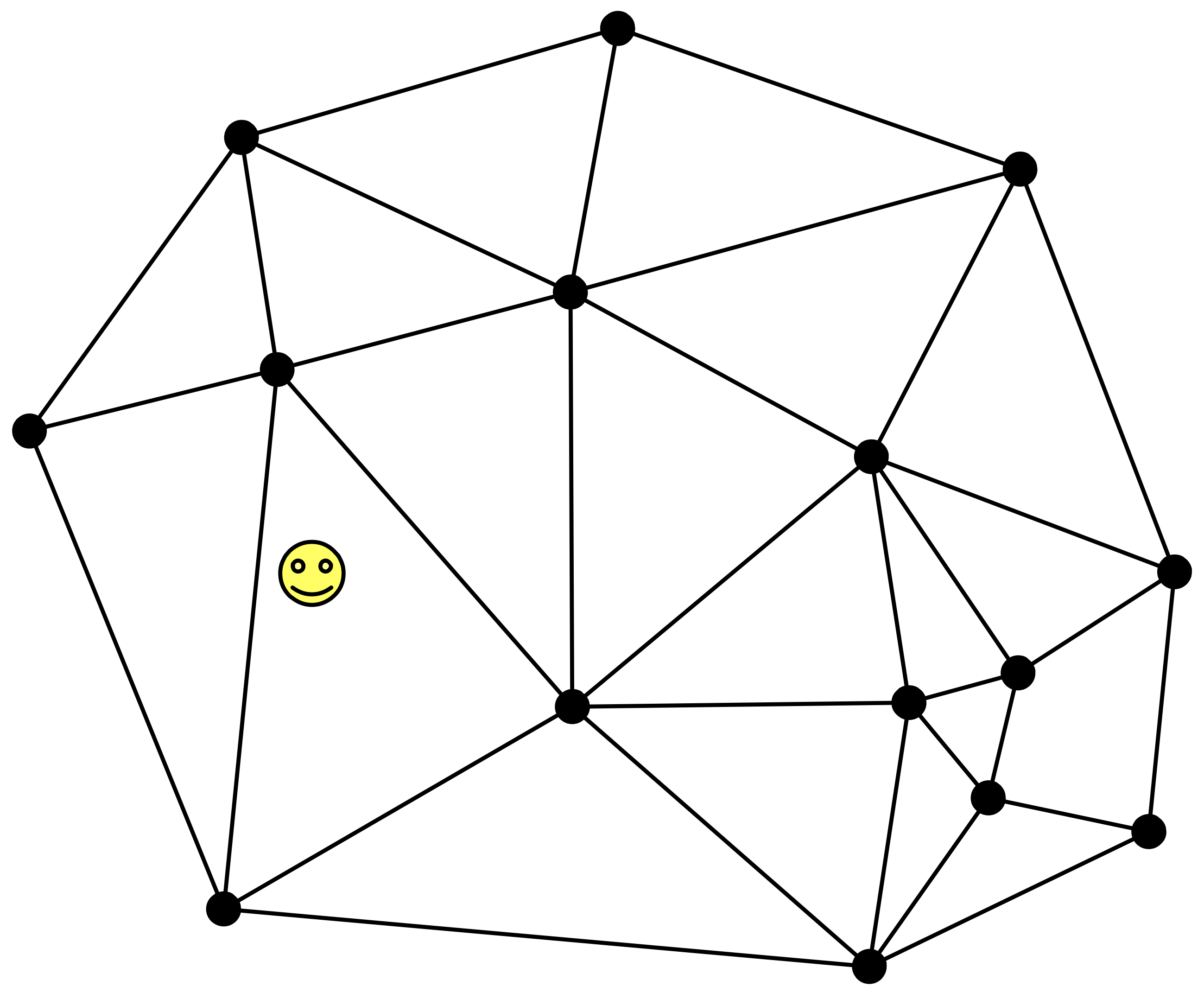


Most of the problems so far stem from the fact, that probes have to placed in a strict way at the grid points.

Let’s assume probes can be placed anywhere instead. We need to decide which probes to take and what weights to use to get an interpolated probe at our character’s position, which is the yellow smiley.

Alternative solution

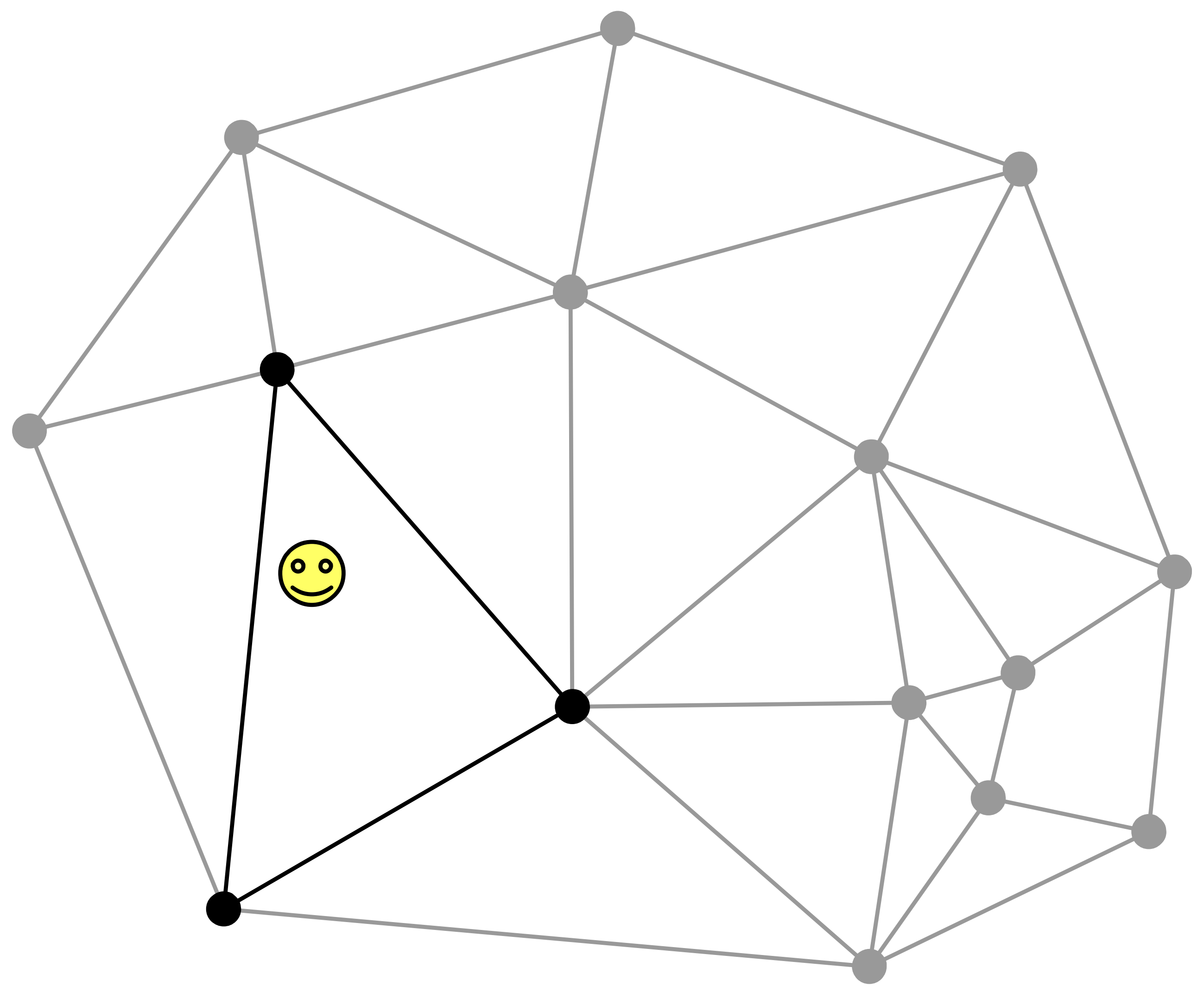
- Delaunay Triangulation maximises the globally minimal angle



Let’s triangulate that set. Delaunay triangulation is the optimal triangulation if we can’t modify the set of points and want to avoid skinny triangles.

Alternative solution

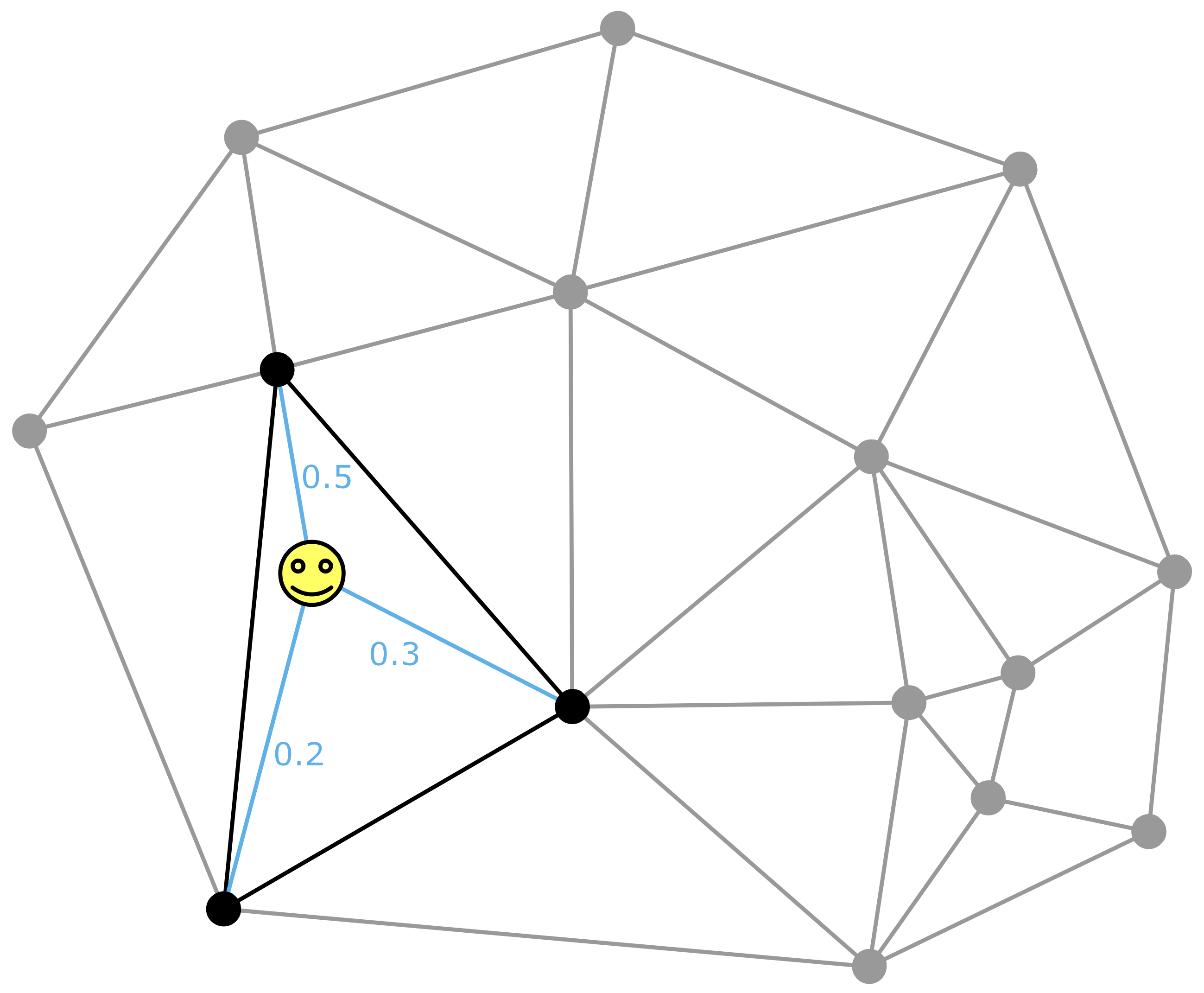
- This triangle contains the position



Let’s say only those 3 probes should influence the character.

Alternative solution

- Barycentric coordinates are the weights



By taking the barycentric coordinates as the weights, we end up with the familiar triangular interpolation.

Triangular interpolation properties

- A good interpolation method is:
  - **smooth** - C0 continuous ("popping" is the worst artefact, we're wired to see it more than anything else)
  - **exact** - when at a sample location, give a weight of 1 to that sample and 0 to all the others, because here we really know the result!
  - **local** - no samples past the closest samples should be blended in
  - and more: monotone, weights $\in[0,1]$ and sum up to 1

Let's step back and think how we would like our interpolation method to behave. We want it to be smooth, otherwise we'll instantly see any popping. We want it to be exact – if we're sampling at a point where we already have a probe, we want that exact probe as a result, no influence from the others. After all that's our ground truth and any other result would actually be a blur.

Thirdly, we only want local probes to be used. Now that's a bit hard to define when density can vary, but it's natural we wouldn't want to use a probe if there's another probe in the same direction, but closer.

Triangular interpolation has all those properties, so that's nice.

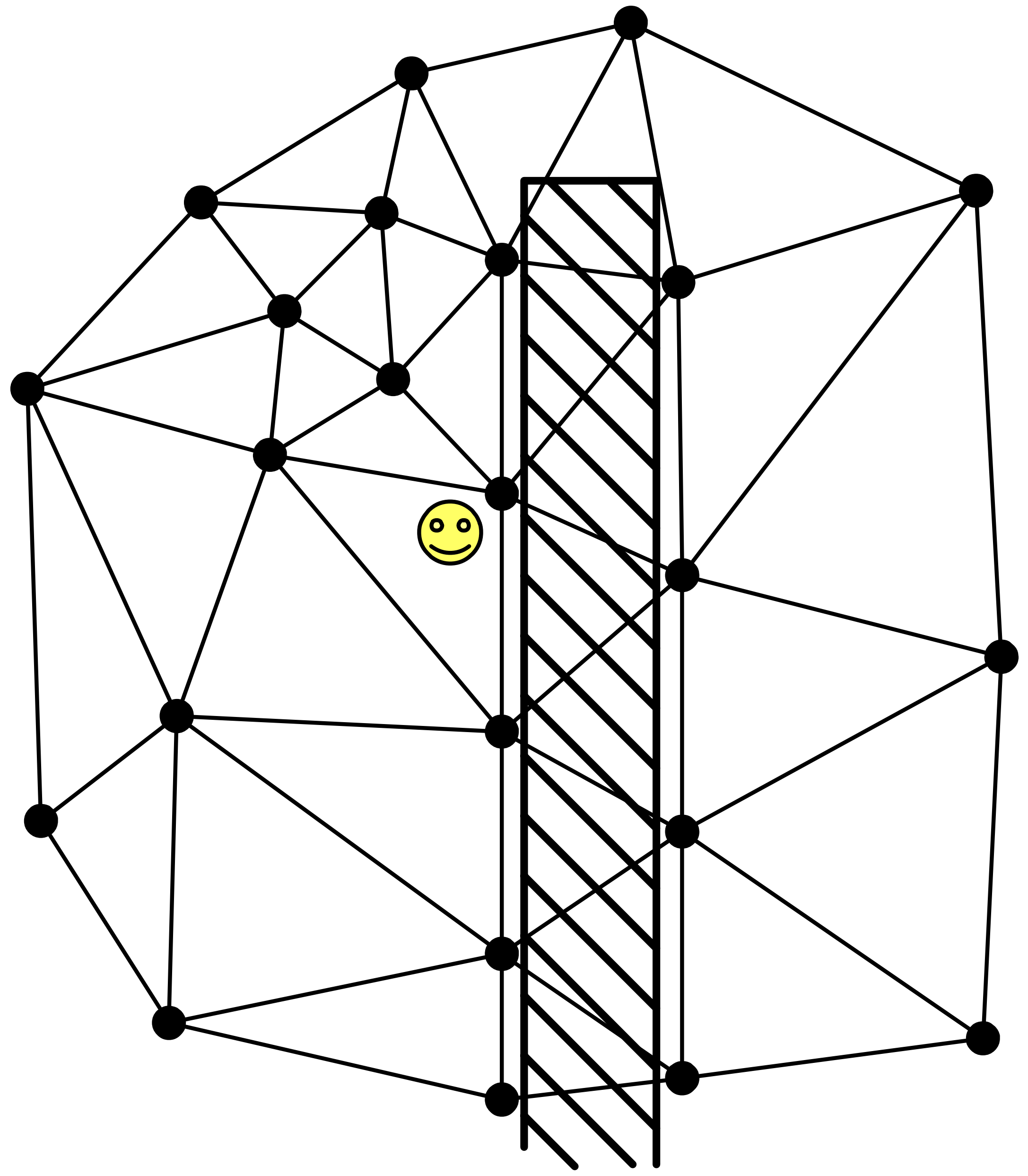
Triangular interpolation

- How does it solve the original issues?
  1. Probes can be placed anywhere
     - Dense samples only at key locations

How does that improve things? Well, now you can place probes anywhere: densely where there are a lot of high frequency lighting changes you want your character to reflect and sparsely elsewhere.

Triangular interpolation

- How does it solve the original issues?
  1. Probes can be placed anywhere
     - Dense samples only at key locations
  2. The interpolation is highly local
     - Limited to the current triangle

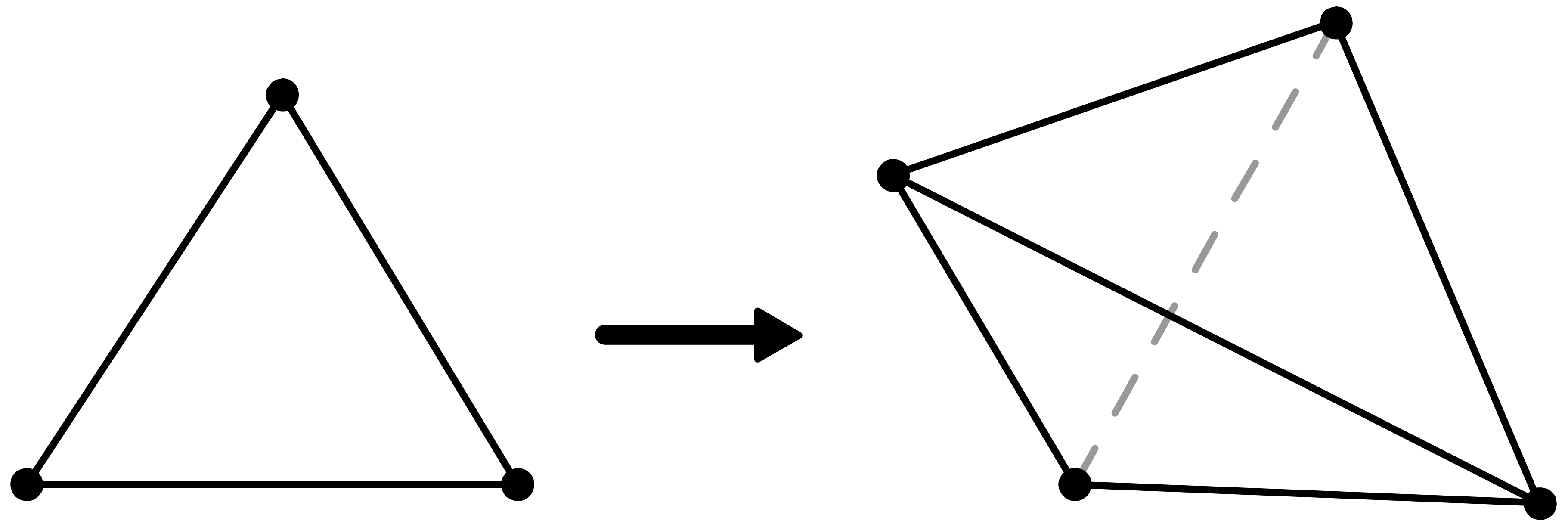


Probes don't end up in walls or obstacles anymore.

Also the unwanted influence of the probes from the right side of the wall can be completely avoided by placing a couple of probes along the wall. Since the triangulation is Delaunay, there will be no long and narrow triangles spanning larger distances – most triangles will be well-shaped. The influence of each probe will be limited to a roughly circular area of around 6 triangles originating from that probe. A probe's influence area can be limited by adding probes around it.

Another way of looking at the problem is to realize that any time the interpolation result is incorrect because of an unwanted probe affecting certain location, there must be a light gradient in between that should be sampled with an additional probe.

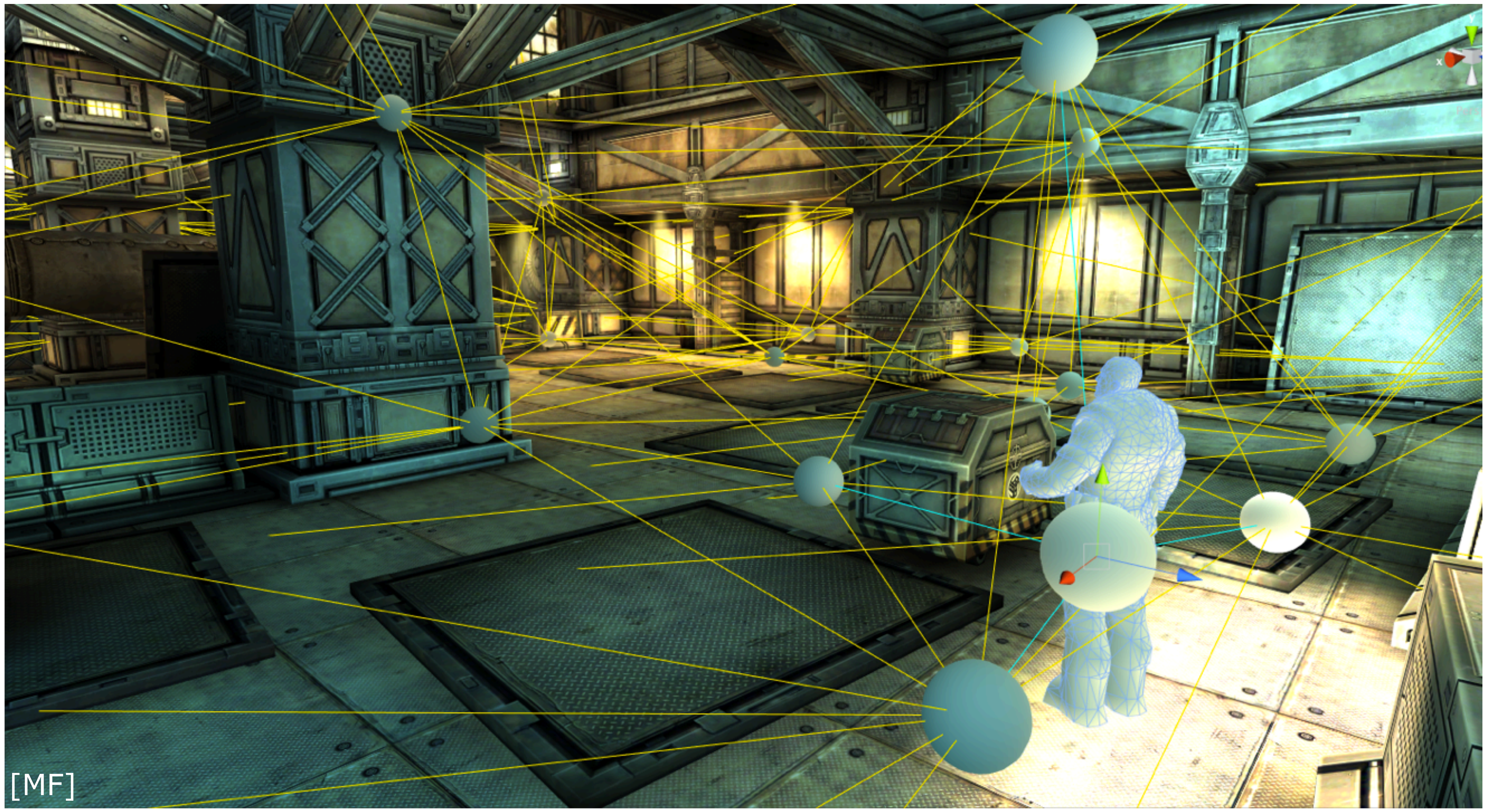
And now for 3D



3D. In 3D the simplices are tetrahedra.

And now for 3D

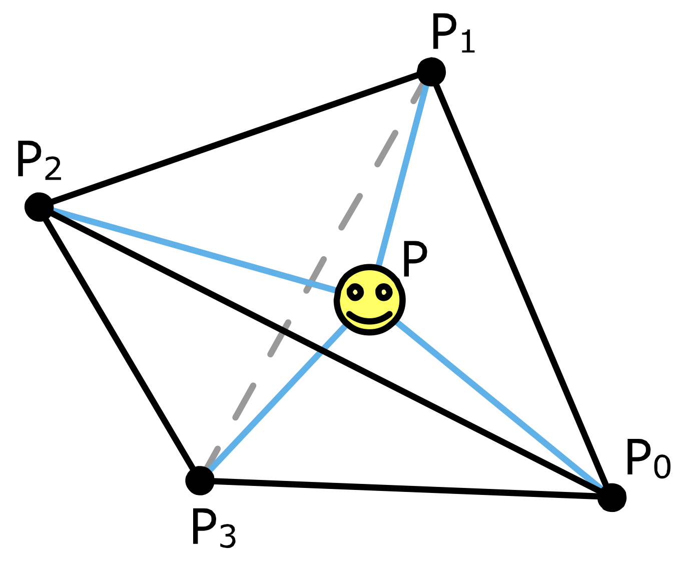
- Delaunay Tetrahedralisation



So instead of a triangular mesh, we use a tetrahedral mesh to subdivide the space.
Again, in our case an optimal tetrahedralisation is the Delaunay Tetrahedralisation.

And now for 3D

- Barycentric coordinates



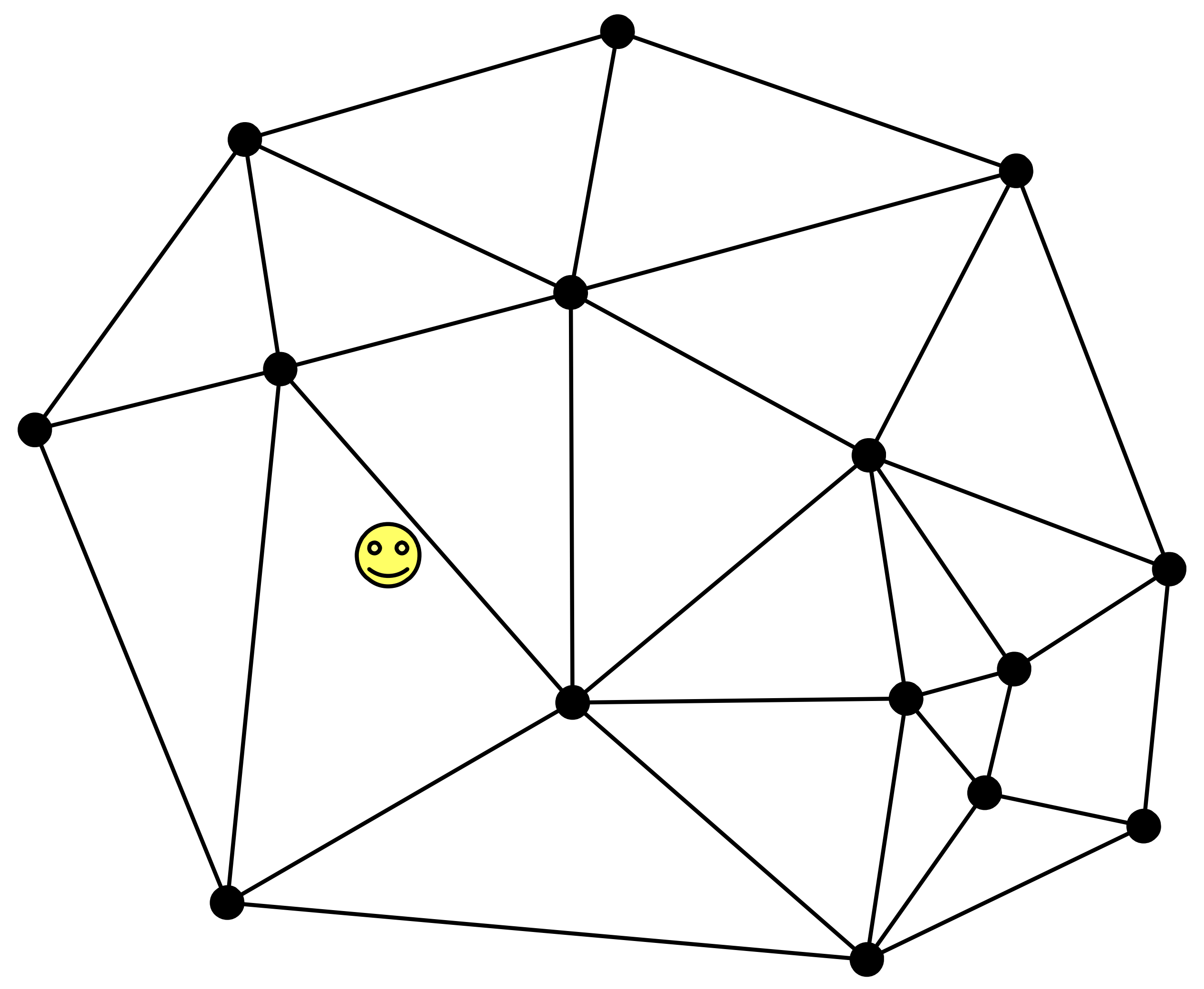
$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \vec{P_0} - \vec{P_3} & \vec{P_1} - \vec{P_3} & \vec{P_2} - \vec{P_3} \end{bmatrix}^{-1} \begin{bmatrix} \vec{P} - \vec{P_3} \end{bmatrix}$$

$$d = 1 - a - b - c$$

Barycentric coordinates in 3D are a natural extension from 2D, with one more coordinate and one more vertex to take into account.
The coordinates are shown in blue. The operation boils down to inverting a 3x3 matrix (that’s the term in the middle). Note that the matrix doesn’t depend on the character’s position P, it depends only on the tetrahedron, so in a way it describes it’s shape. We’ll use that fact later.

Search

- Which tetrahedron contains the object?

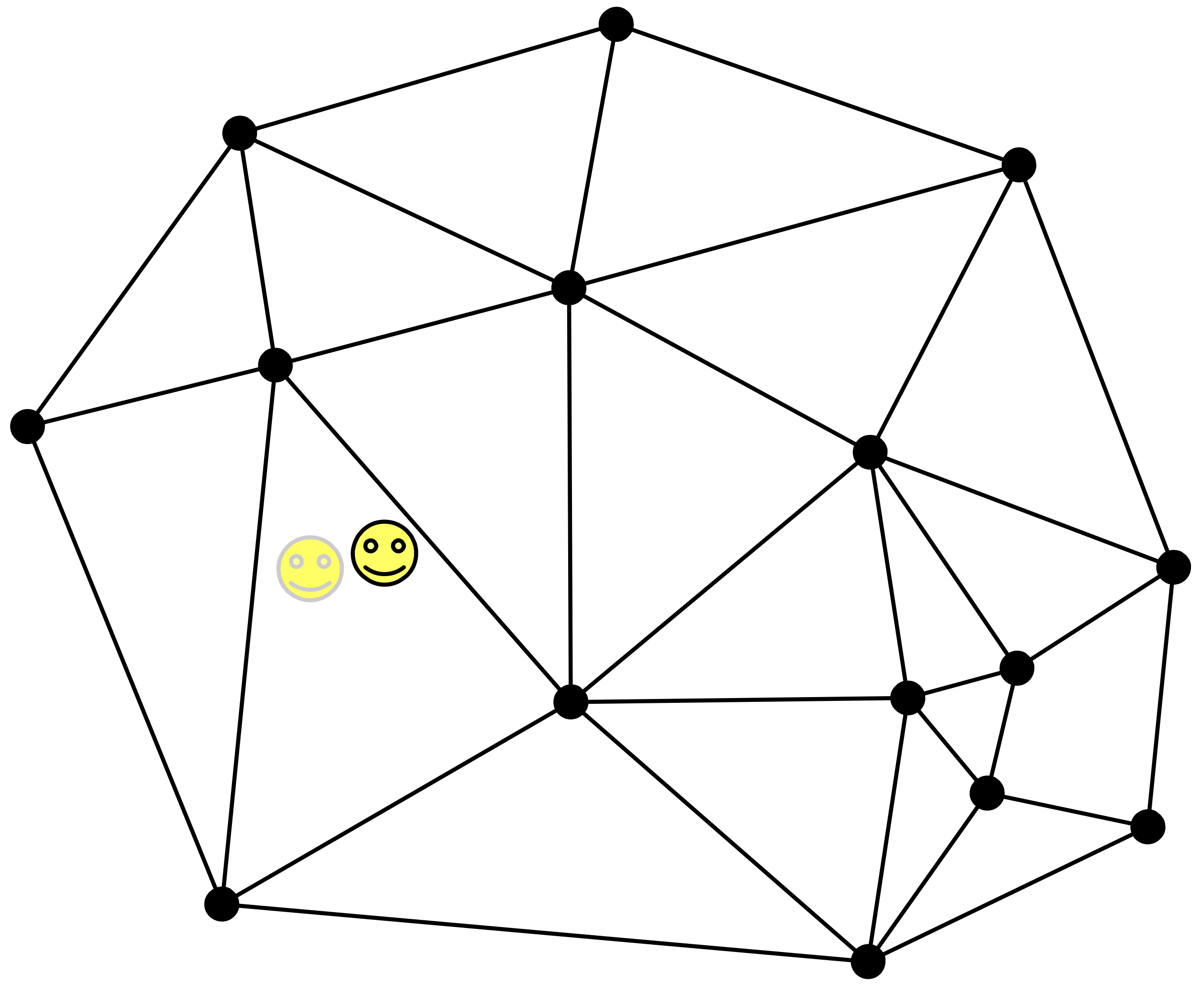


Now that we know how to calculate the weights once inside a tetrahedron, we need a way to find out which tetrahedron we're in.

The images are still showing triangular meshes for simplicity, but let's imagine these are tetrahedra – the analogy works well.

Search

- Objects cache the tetrahedron index from the previous frame

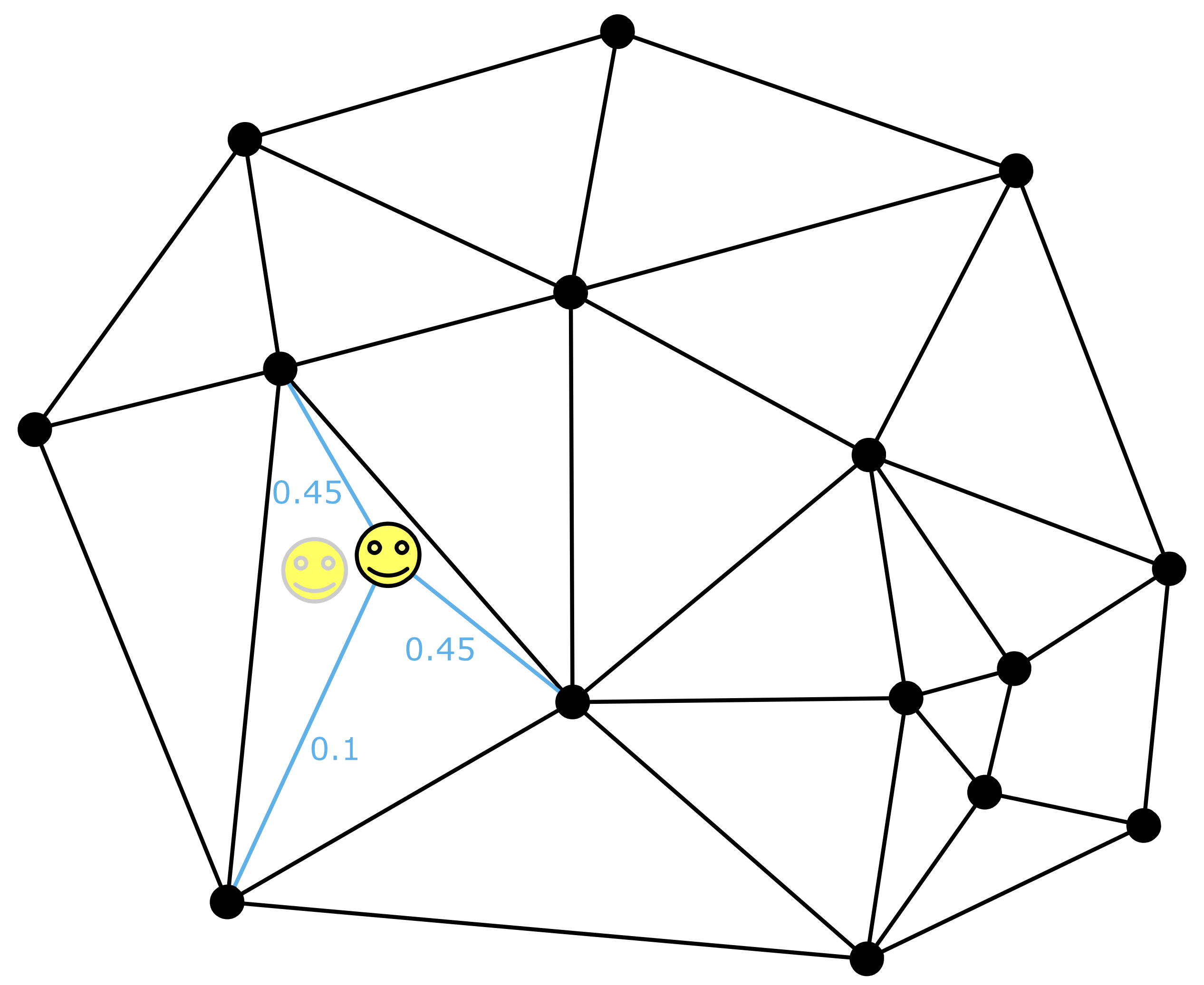


- Checking barycentric coordinates...

We know in which tetrahedron we were inside in the last frame, since we cache that information. Let’s assume we’re still in the same one and calculate barycentric coordinates based on that assumption.

Search

- All coordinates positive, we're still inside

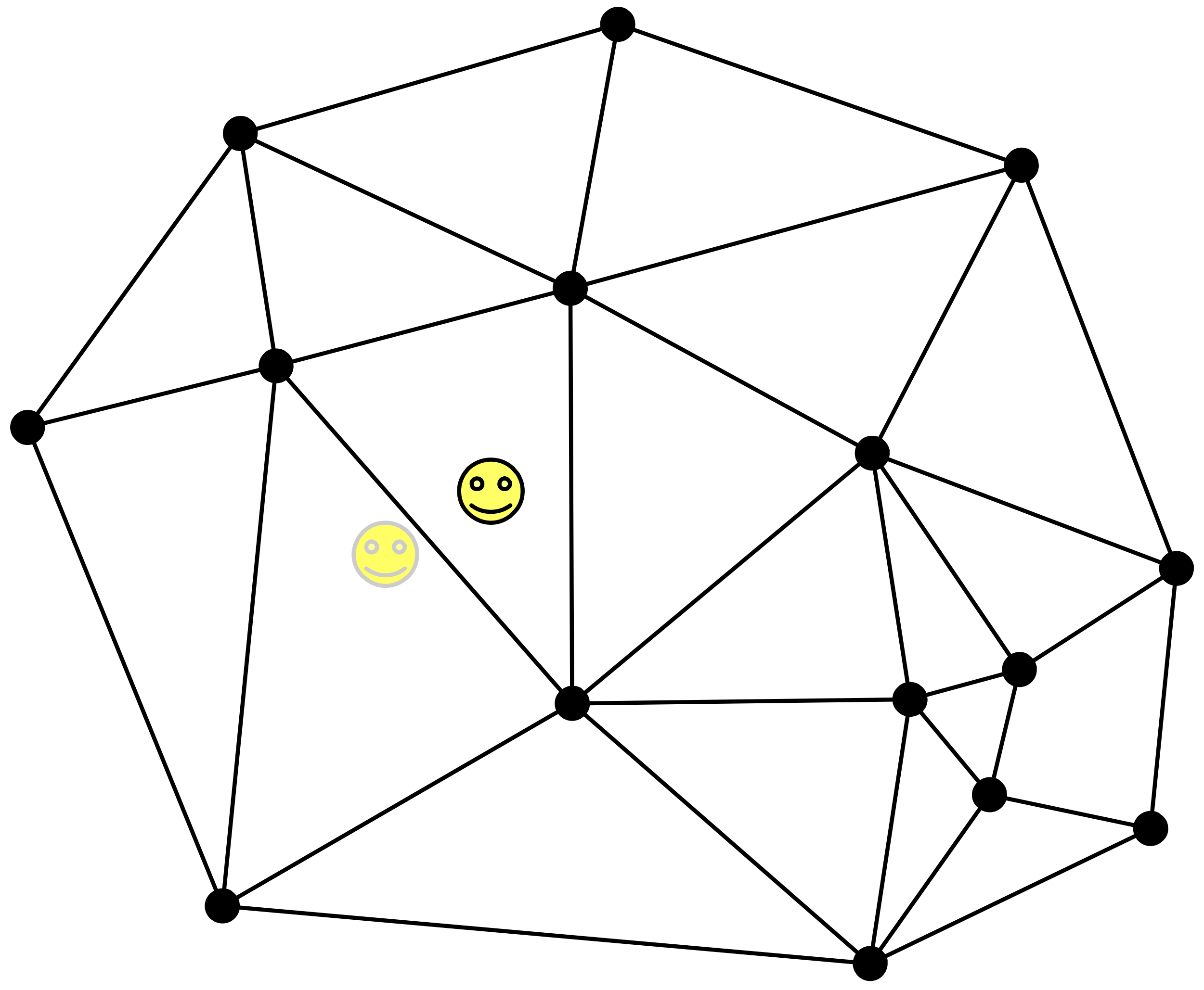


- The calculated coords used directly as probe weights

Turns out all barycentric coordinates are positive, which means we're still in the same tetrahedron. As a bonus what we just calculated can be directly used as probe weights.

Search

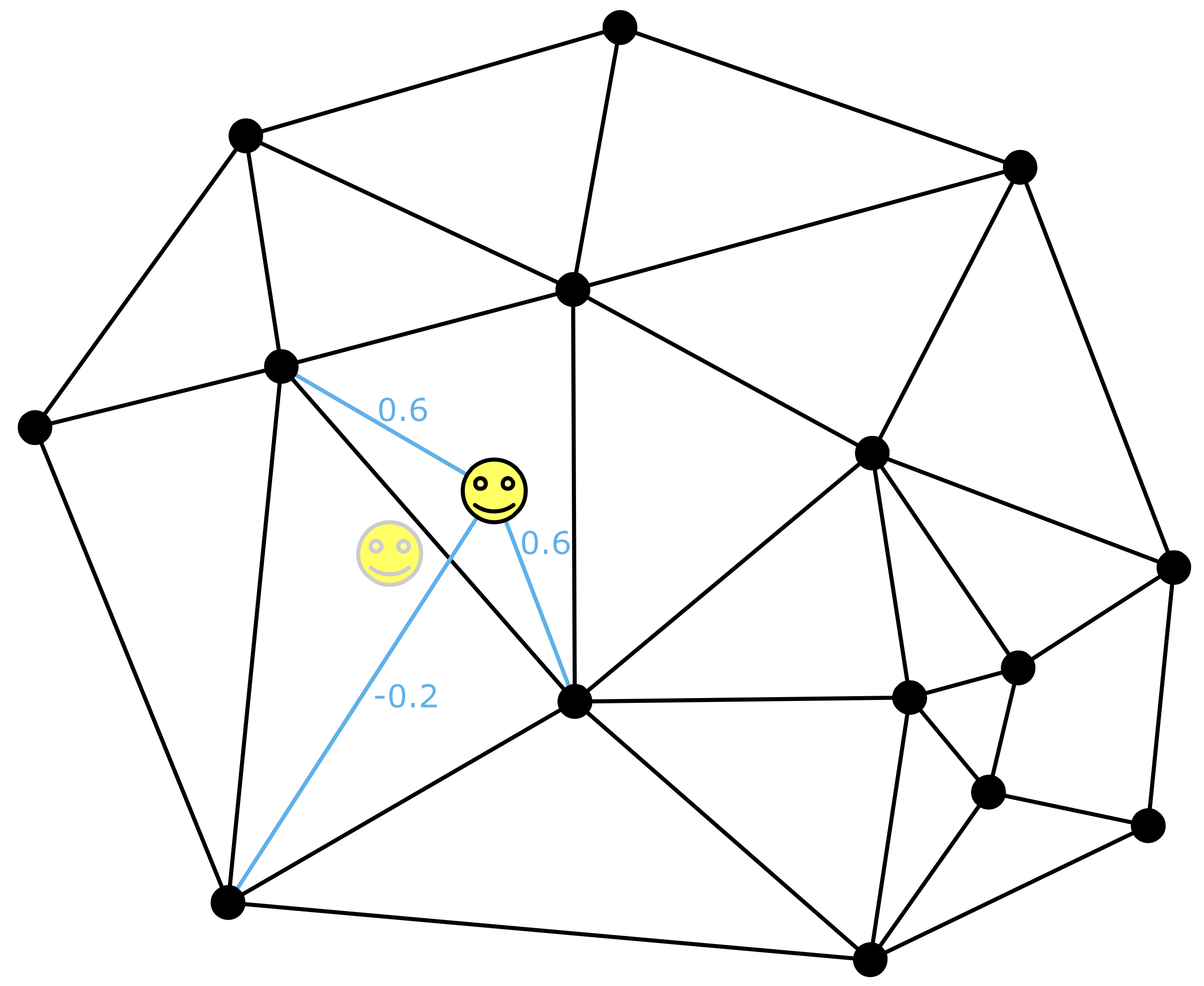
- Next frame, the object moved



Next frame we’re not so lucky: the object moved to a different tetrahedron, but we don’t know that yet.

Search

- Try the last tetrahedron

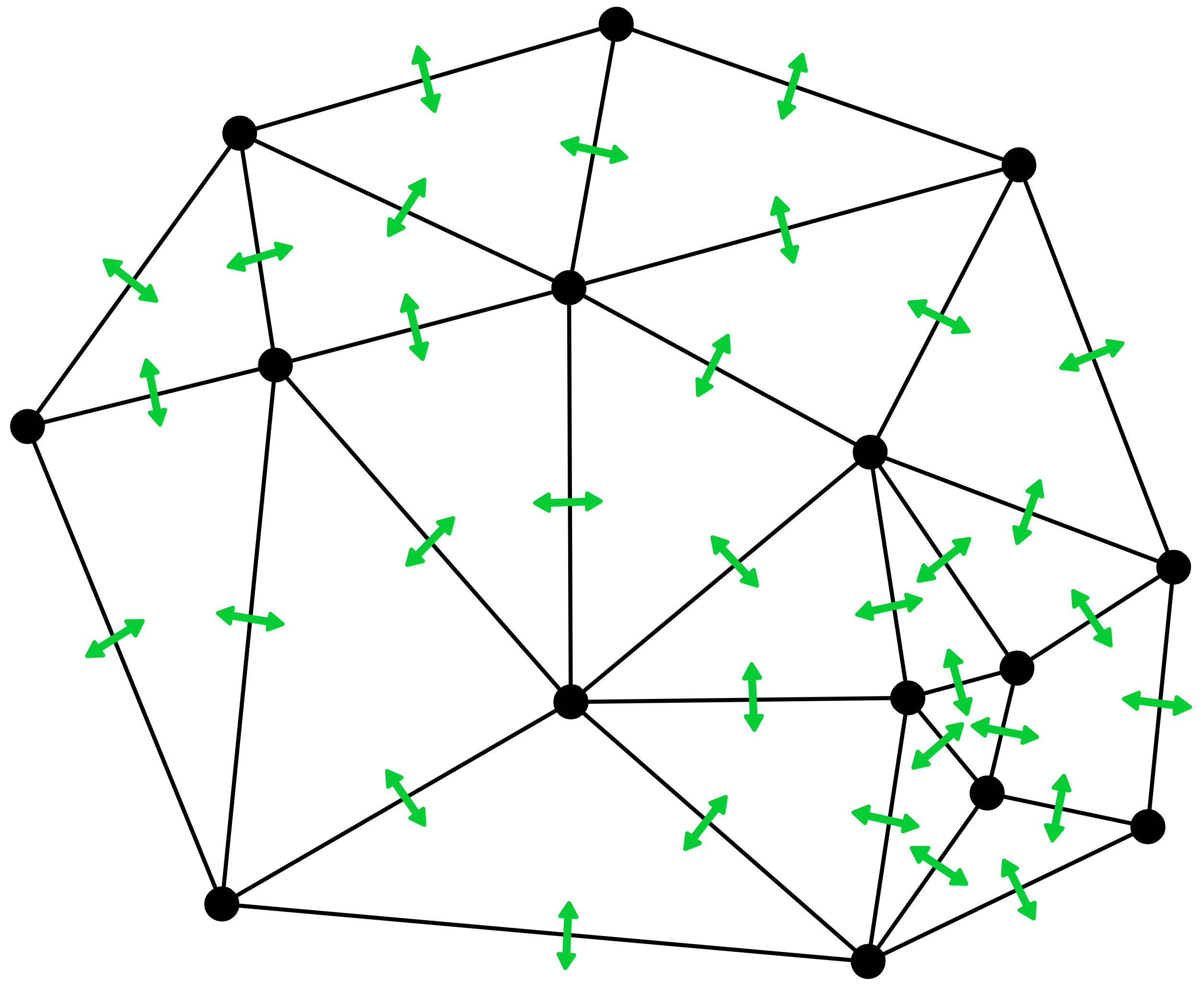


- One of the coords negative, we’re outside

We proceed as before: assume we are in the same tetrahedron as in the last frame. This time one of the coordinates is negative, which gives us a hint we’re outside.

Search

- Adjacency graph

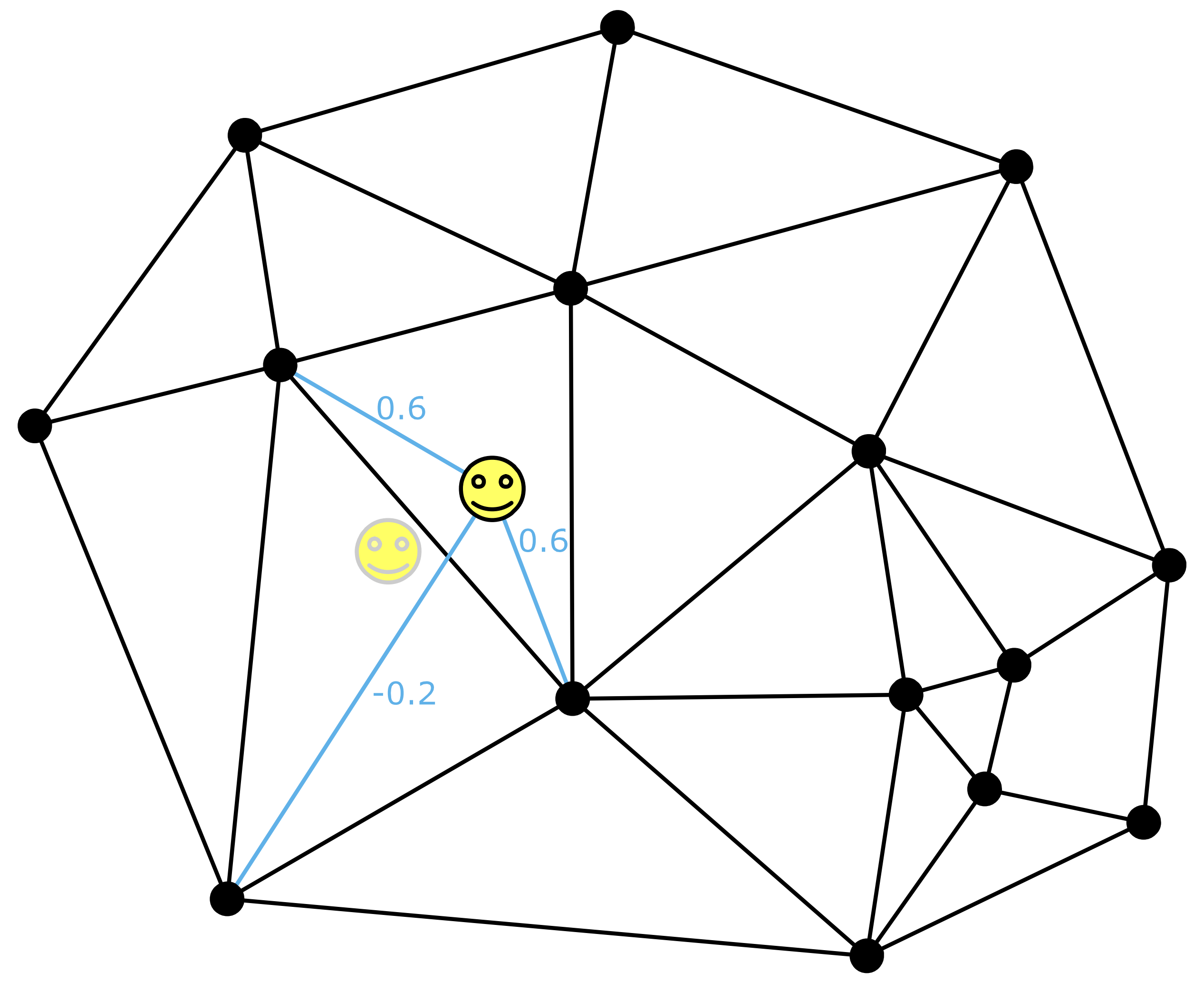


- Each tetrahedron stores indices of it’s 4 neighbours

At this point we need some additional topology information: each tetrahedron has exactly 4 neighbours and we need to know their indices.

Search

- Where to next?

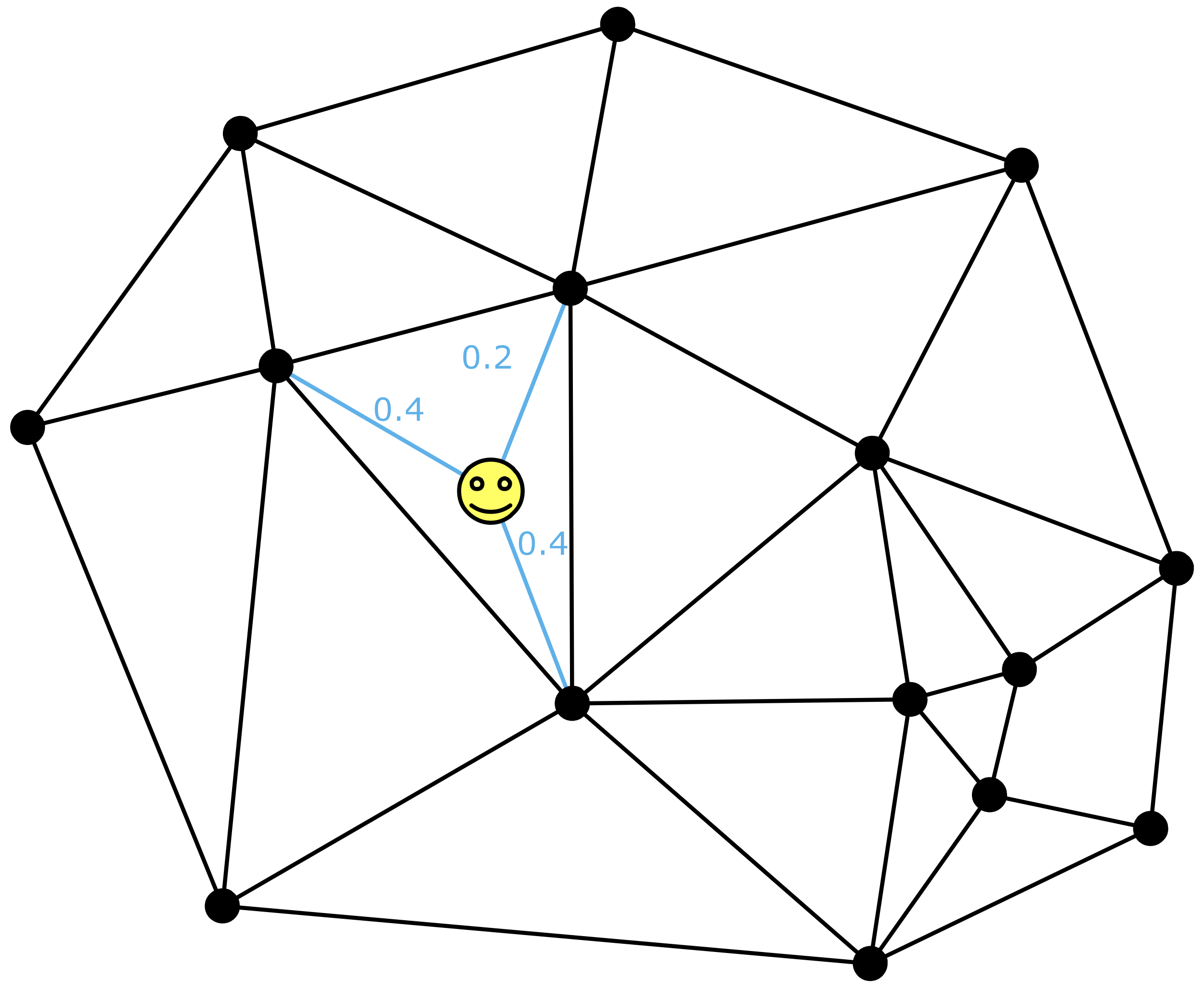


- Towards the neighbour at the face opposite to the most negative barycentric coord

To decide which neighbour we should test next, we look for the most negative coordinate and move in the opposite direction.

Search

- Success!

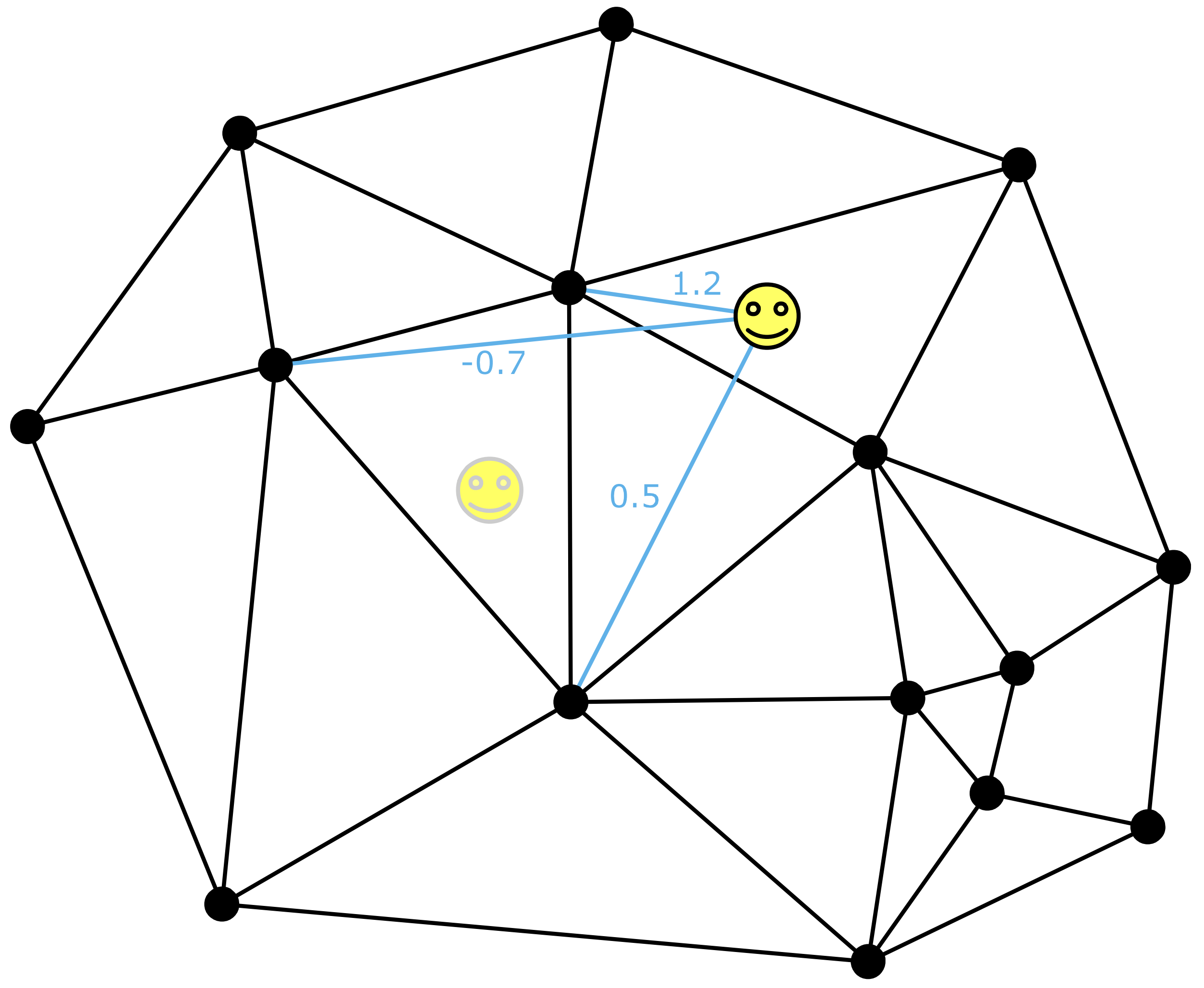


- Update the *last frame tetrahedron index*

We test this tetrahedron now, and it’s a hit. We update the cache as the current tetrahedron will be our new best guess in the next frame.

Search

- Next frame, the object moved even more

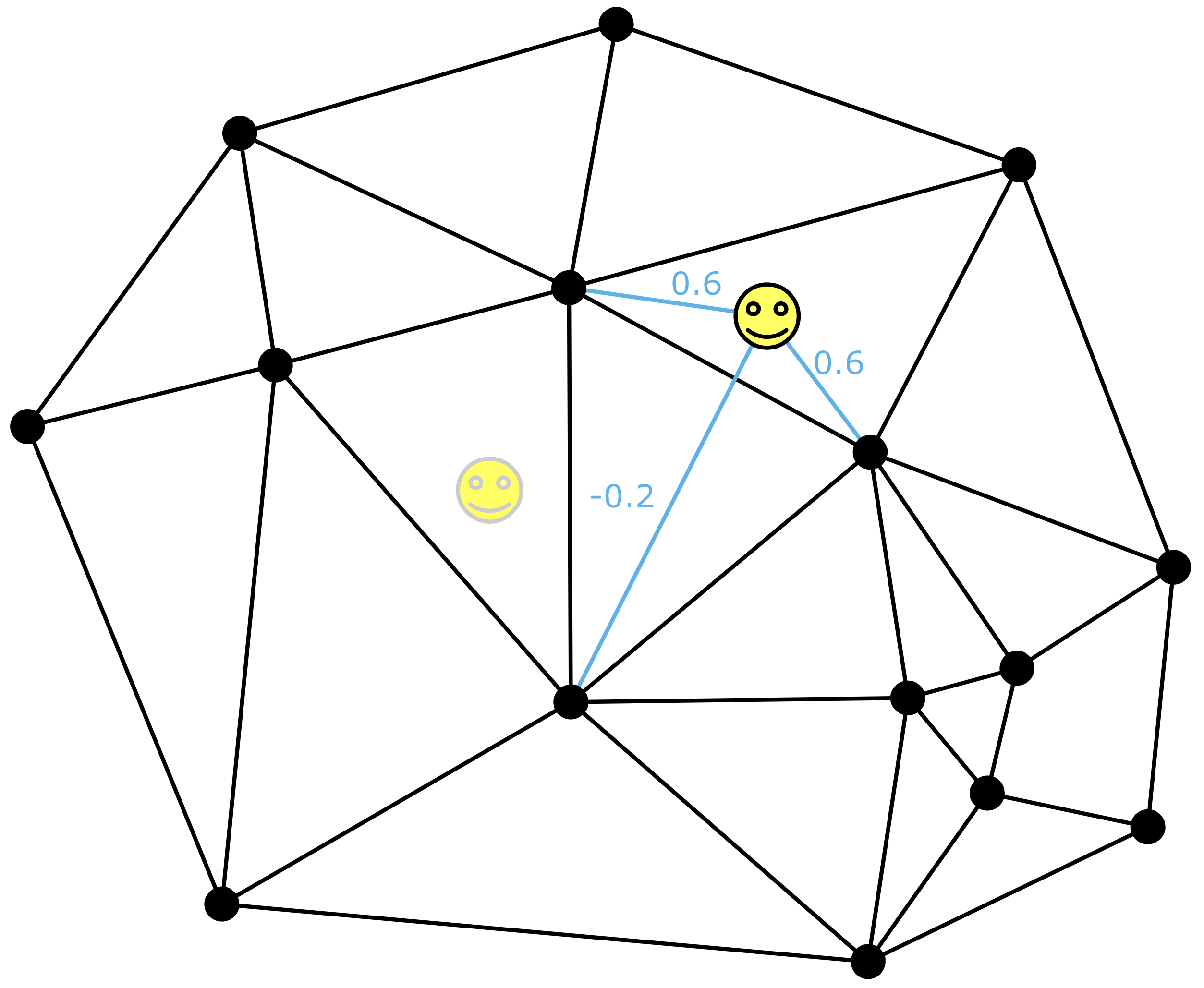


- Towards the neighbour at the face opposite to the most negative barycentric coord

Let's try this again. Now we moved even more. We test our best guess, but one of the coordinates is negative. We move to the neighbour on the right.

Search

- Next frame, the object moved even more

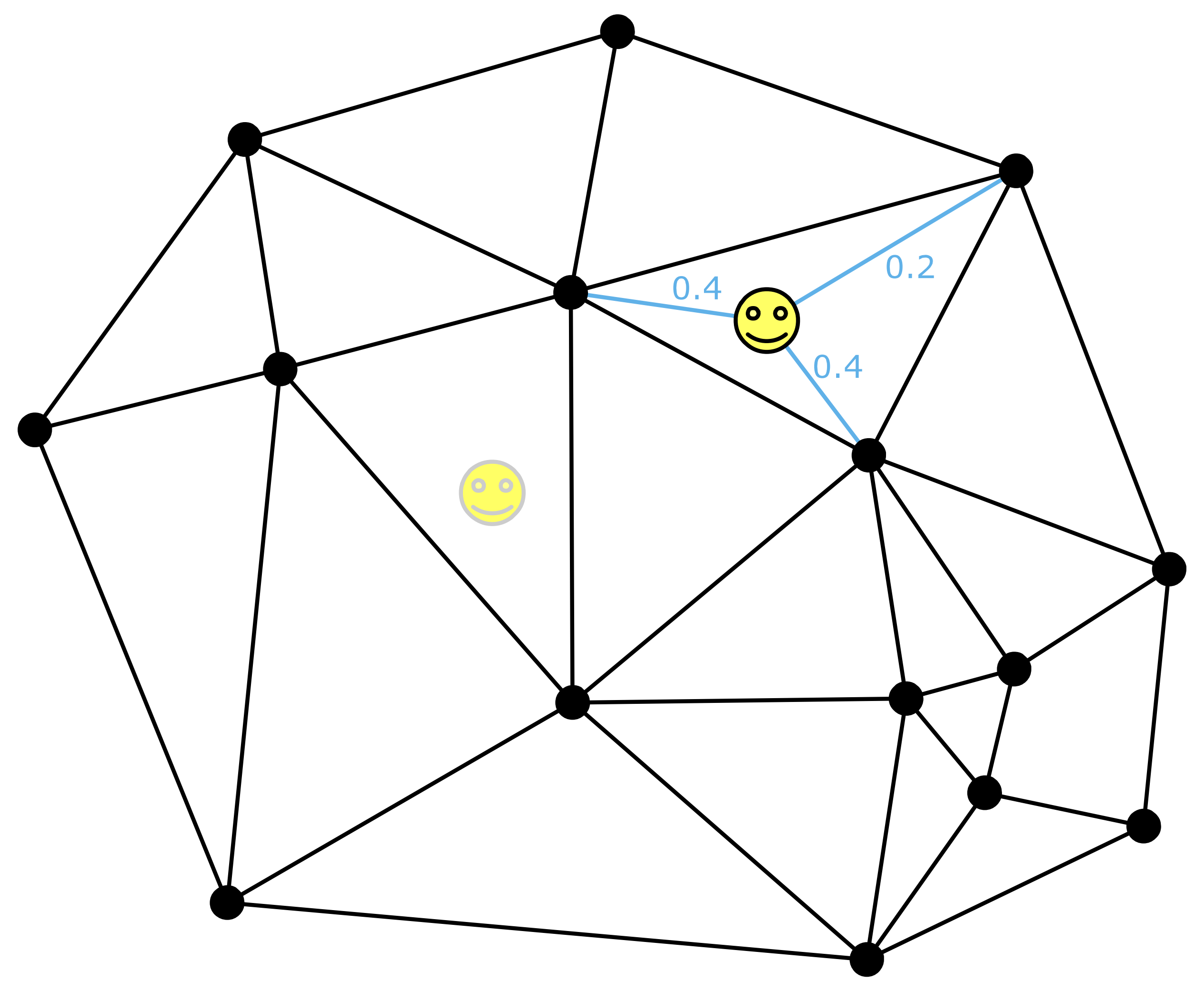


- Towards the neighbour at the face opposite to the most negative barycentric coord

The neighbour returns a negative coordinate, so we move to the next one, again towards the most negative coordinate.

Search

- Success!



It's a hit.

Search

- An intuitive thing to do would be to compare dot products with each face’s normals
- Turns out comparing barycentric coords is equivalent and we have them anyway!

To know which direction to move towards we would typically compare dot product of the relative position with each of the face normals and pick the one with the highest result.

But we have already calculated barycentric coordinates and choosing the most negative one is actually equivalent.

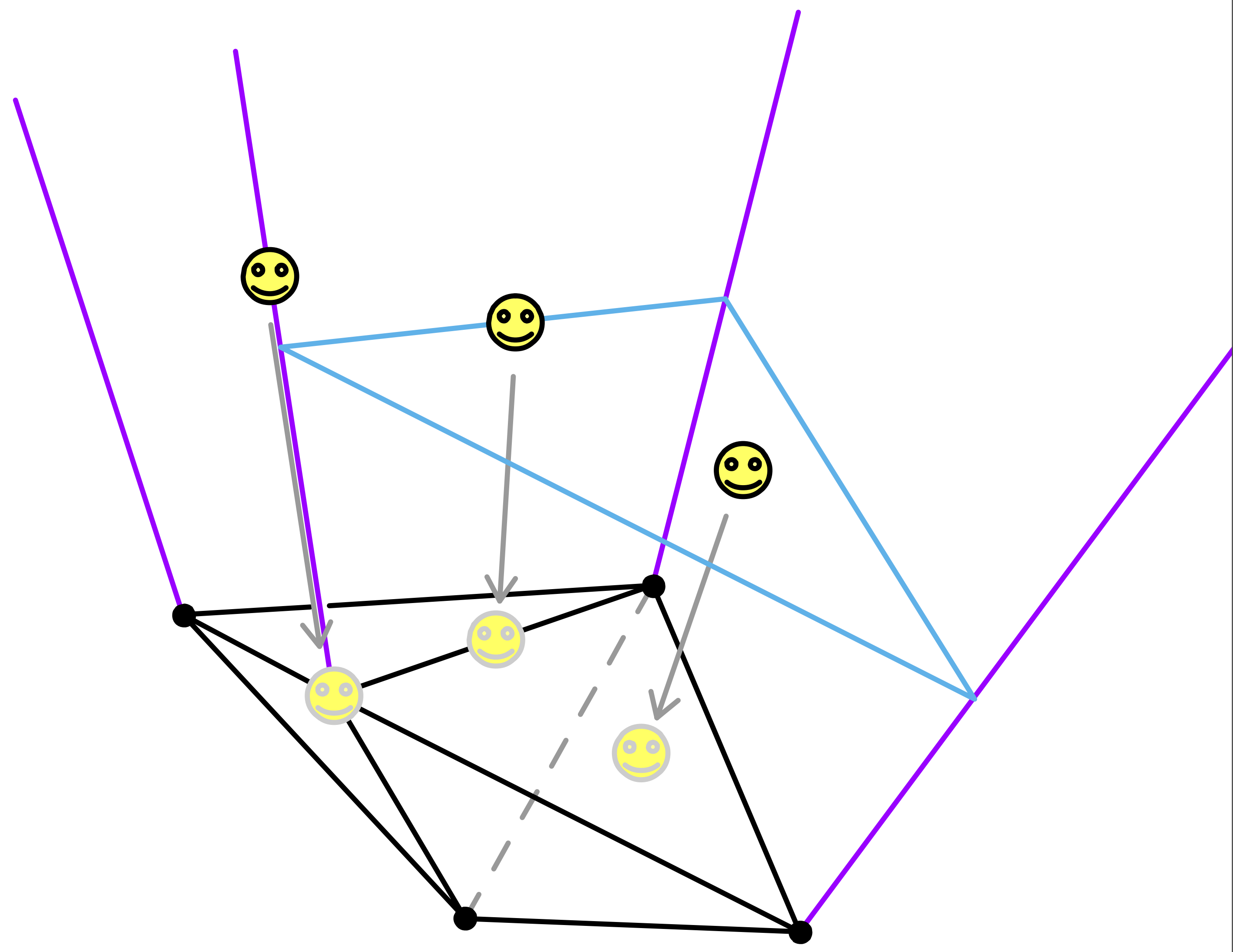
Search

- When an object gets instantiated, it doesn’t have a good tetrahedron index guess
  - We start from 0
  - Tetrahedron 0 should be in the centre
  - Ideally: from which the distance to any other tetrahedron along the adjacency graph is the shortest
  - Decent approximation: at the average probe position

We need a decent initial tetrahedron guess for objects that just got instantiated and we don’t know much about their position relative to the tetrahedral mesh. But if we make sure that the tetrahedron 0 is at the centre (for some definition of ‘centre’), on average we’ll find the correct tetrahedron with fewest steps.

Extrapolation

- What do we do about objects outside of the convex hull?
  - Subdivide the outer space into open cells
  - Within a cell project the position onto the hull face
  - Once on the face, barycentric coords
  - Projection needs to be continuous between the cells



We still need to handle the positions that fall outside of the hull of the probes, ideally by smooth extrapolation of the probes that form the hull.

We can subdivide all of the outer space with cells that are extruded faces of the hull. Within each cell, we need a way to project any position back onto the corresponding face of the hull, since we already know how to handle a point within a triangle.

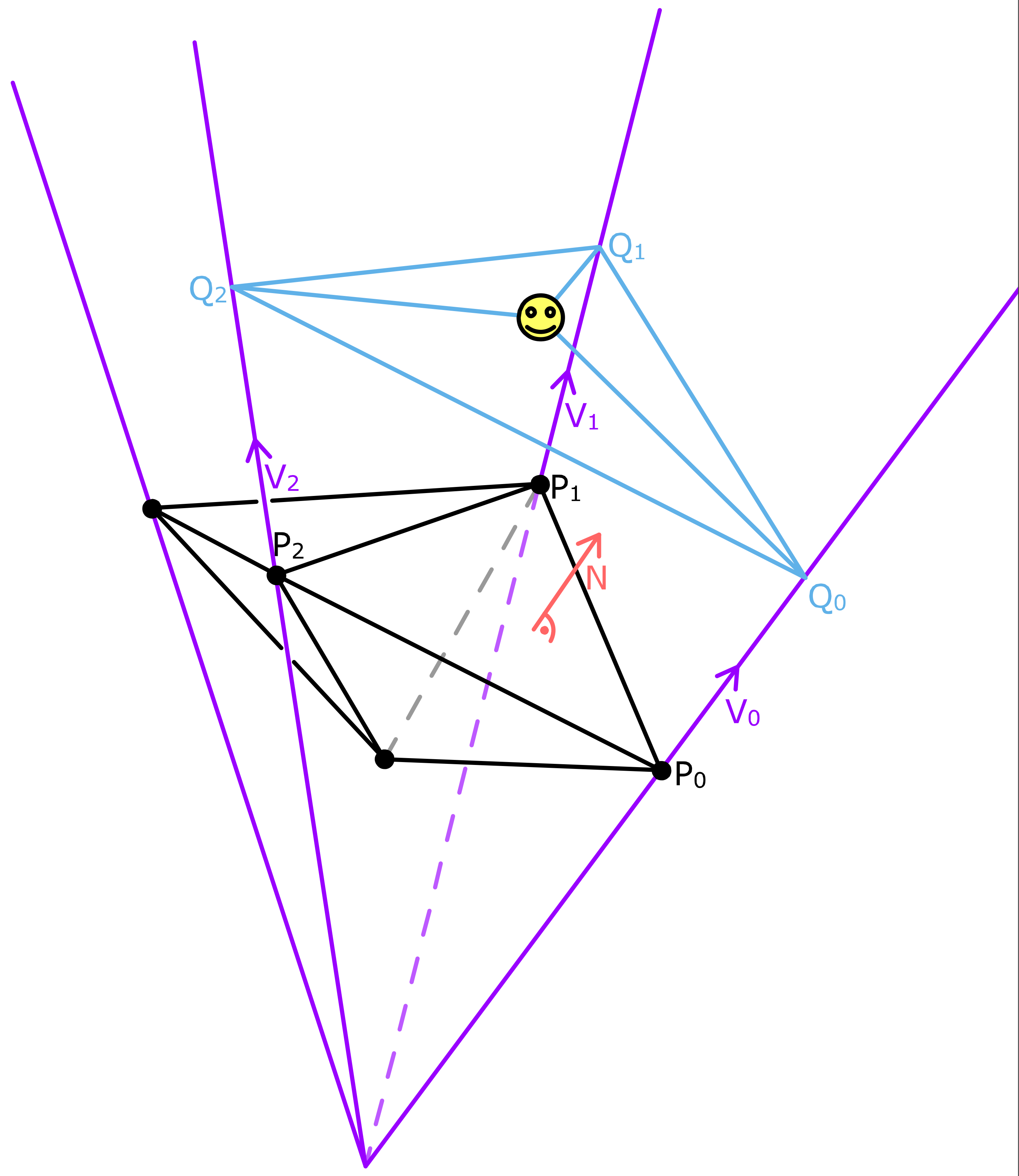
The hard part is making sure the projection is continuous as we go from one outer cell to the other, passing over a hull edge.

Extrapolation

- An easy subdivision
  - All hull rays intersect in the centre
  - Prepare $\vec{N}\cdot\vec{V_0}=1, \vec{N}\cdot\vec{V_1}=1, \vec{N}\cdot\vec{V_2}=1$
  - Find the triangle $t=\vec{N}\cdot(\vec{P}-\vec{P_0})$

$$\vec{Q_0}=\vec{P_0}+t\vec{V_0}$$
$$\vec{Q_1}=\vec{P_1}+t\vec{V_1}$$
$$\vec{Q_2}=\vec{P_2}+t\vec{V_2}$$

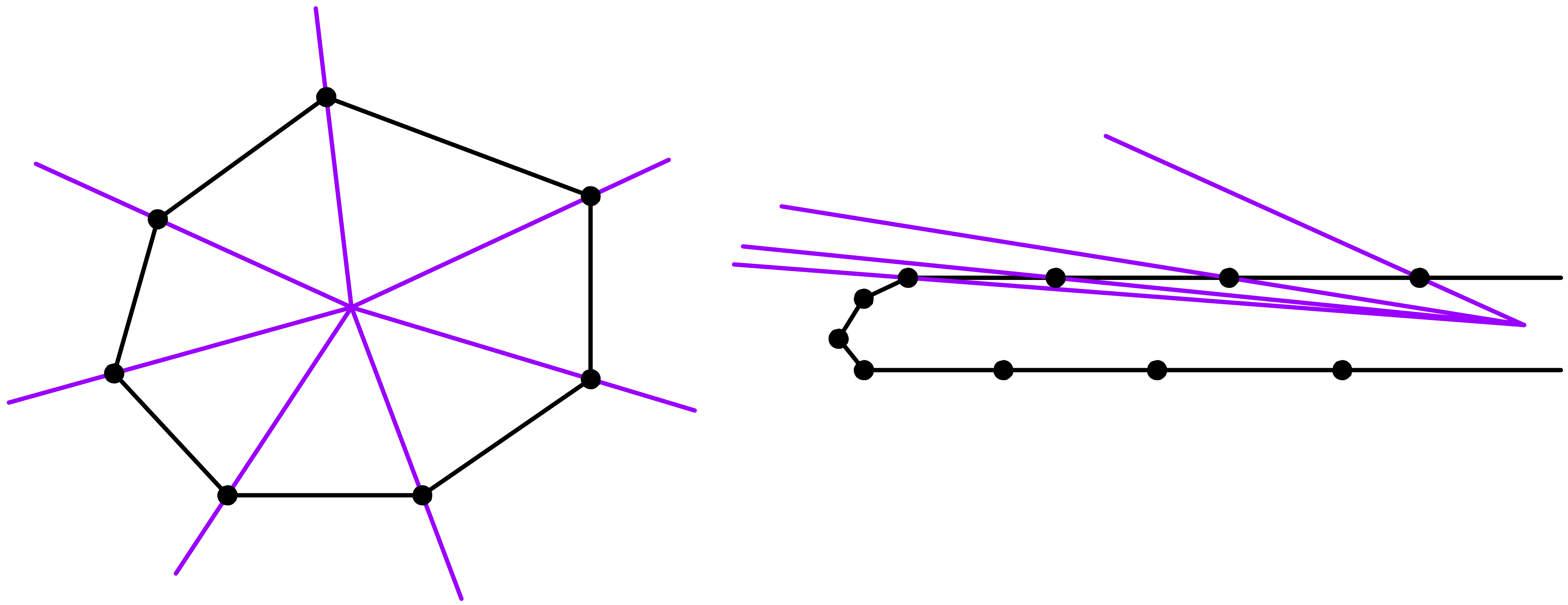
  - Once we have the triangle, calculate barycentric coords (robust implementation in [RTCD])



One solution is to form the outer cells by extruding the hull faces along rays, which all meet at the centre of the entire mesh. The math is very easy in that case, but there's a problem.

Extrapolation

- The easy subdivision is not universal enough
    - Doesn't really work for flat probe sets

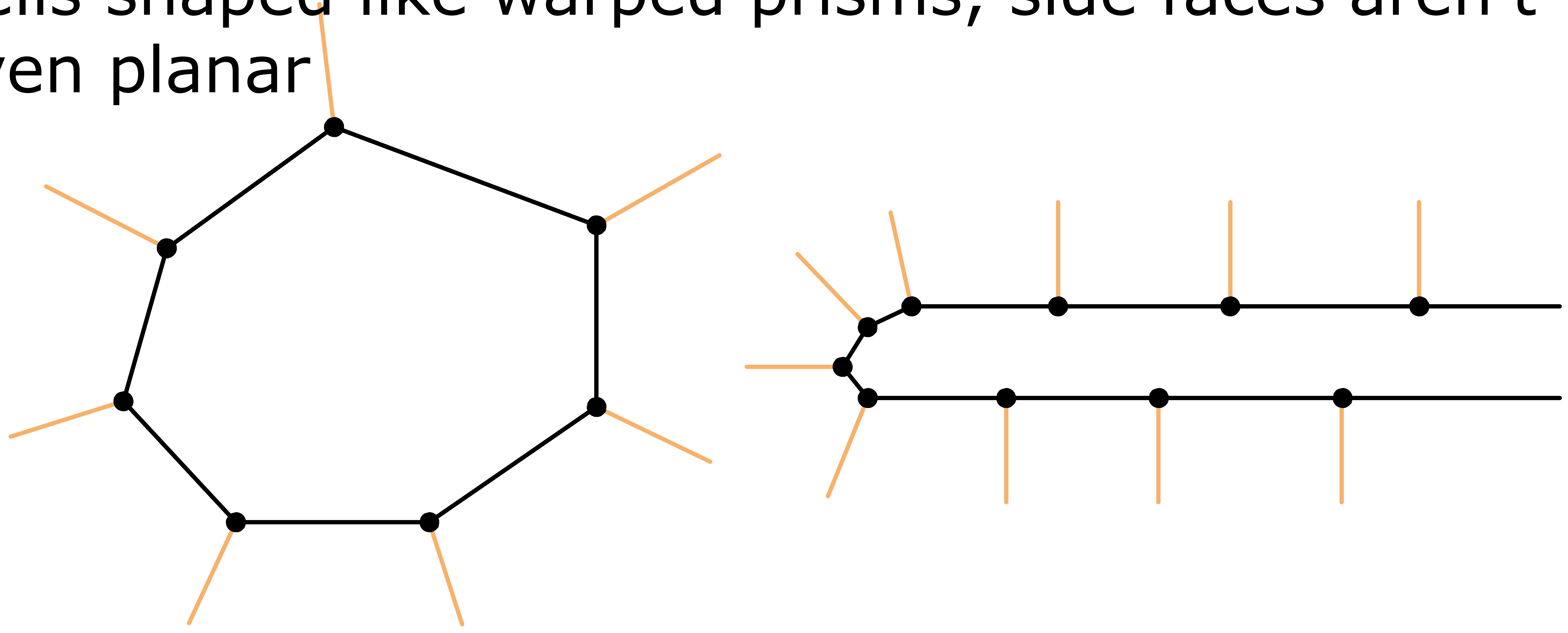


The requirement of all rays intersecting at one position leads to badly-shaped outer cells if the probe set happens to be not very round.

Outer cells start running along the surface of the hull. As we move away from the hull, we cross multiple thin cells, which results in unwanted quick changes in illumination and generally bad extrapolation.

Extrapolation

- Cells based on proper hull vertex normals
  - Projection will be tricky
  - Cells shaped like warped prisms, side faces aren't even planar



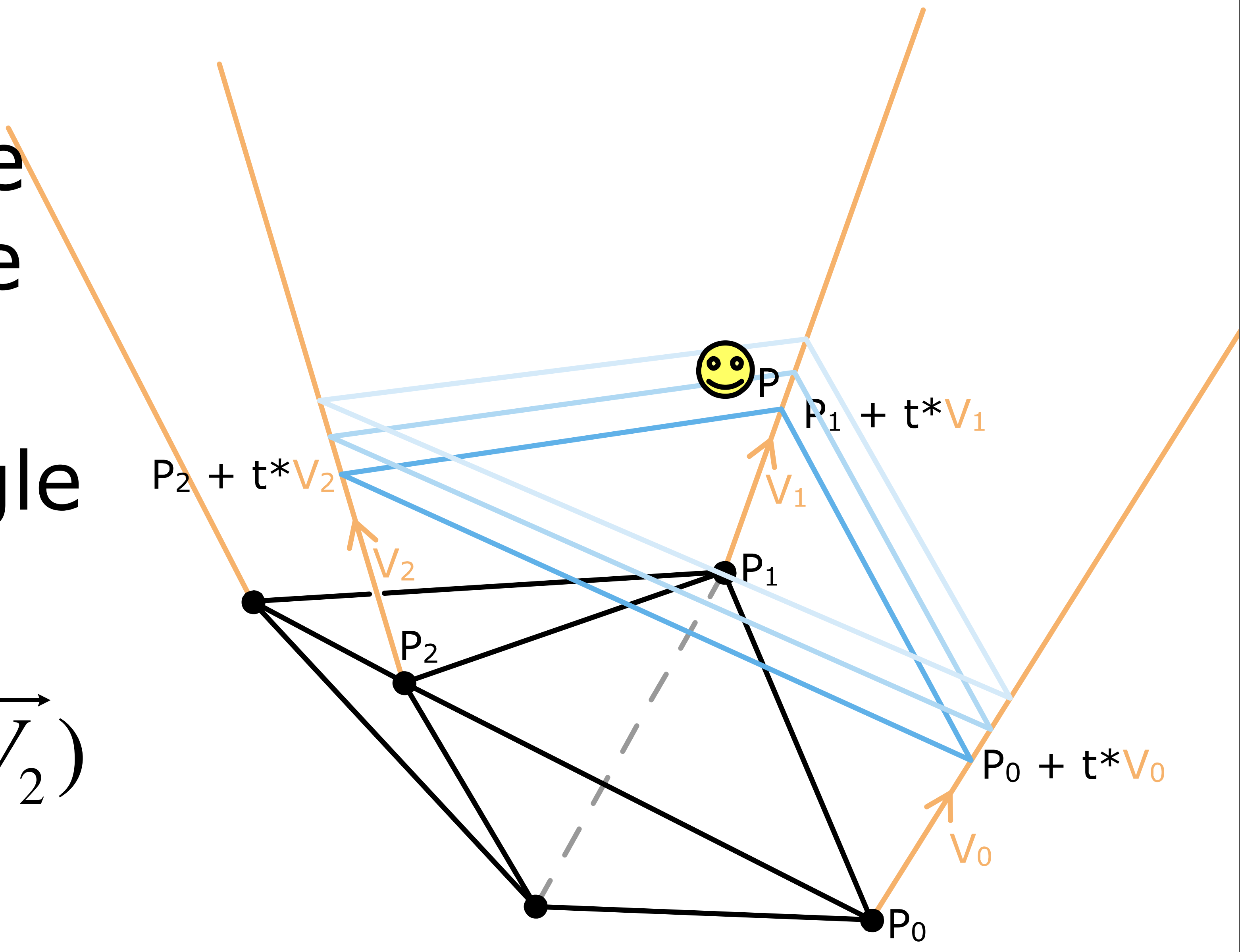
Ideally the outer cells should be roughly perpendicular to the surface. We can do that by spanning the outer cells between the vertex normals. In 3D the problem is that adjacent normals aren't necessarily coplanar, so the resulting shapes will be those twisted or warped prisms, but let's try to deal with that.

Extrapolation

- As t goes from 0 to ∞, the triangle sweeps through the entire volume of the cell
- For exactly one t value, the triangle will contain the object at P

$$\vec{P} = a(\vec{P_0} + t\vec{V_0}) + b(\vec{P_1} + t\vec{V_1}) + c(\vec{P_2} + t\vec{V_2})$$

$$c = 1 - a - b$$



If we look at the image, P0, P1, P2 is an outer face. The rays extending from each of those 3 vertices are forming the outer cell, which extends to infinity. V0, V1, V2 are the direction vectors of those rays. To get any point on the ray 0, we just take P0 + t*V0, where t goes from 0 to infinity.

If we use the same t for all 3 rays, we will get a triangle that sweeps through the entire volume of the outer cell – starting at the face and going upwards to infinity. For some value of t the triangle will contain the object's position at P (the smiley).

At this point the triangle will be coplanar with P, which means P can be represented as a linear combination of the triangle's vertices. a, b, c are then the barycentric coordinates.

Extrapolation

- To solve for t, a and b
  - Rewrite as: $a(\vec{A}+t\vec{A'})+b(\vec{B}+t\vec{B'})+\vec{C}+t\vec{C'}=0$
  - In matrix form:

$$\underbrace{\begin{bmatrix} \vec{A}+t\vec{A'} & \vec{B}+t\vec{B'} & \vec{C}+t\vec{C'} \end{bmatrix}}_{T}\begin{bmatrix} a \\ b \\ 1 \end{bmatrix}=\begin{bmatrix}\vec{0}\end{bmatrix}$$

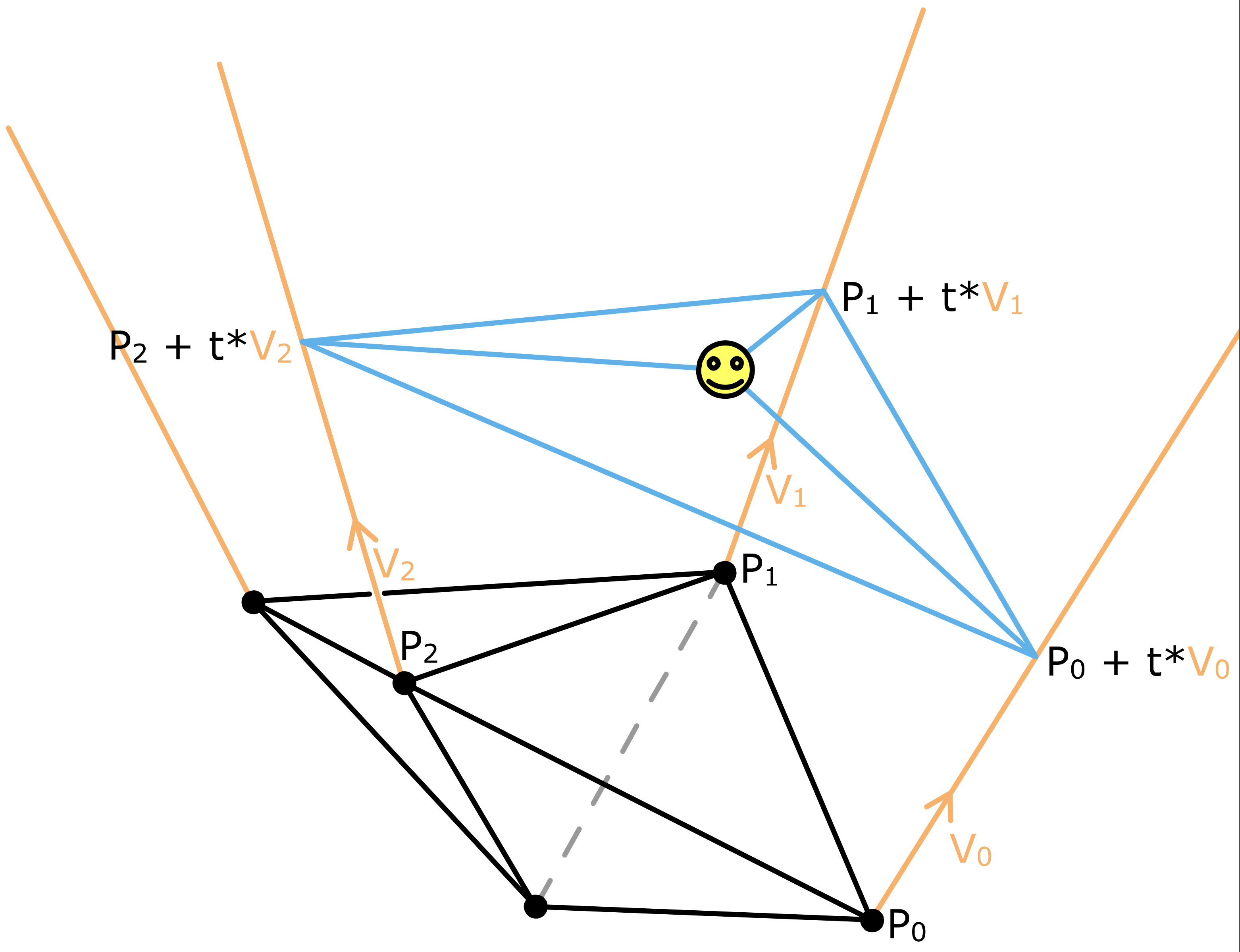
  - T cannot be invertible, so $\det(T)=0$
  - $\det(T)=0$ is a cubic in t: $pt^3+qt^2+rt+s=0$
  - We know it should have exactly one positive root

To solve the equation for t, we can rewrite it in a matrix form where T is a 3x3 matrix. We then notice that T cannot be invertible. If it was, we could multiply the equation by the inverse of T and get [a b 1] equals zero vector, which is false.

If T is not invertible, it’s determinant is 0. T is a 3x3 matrix with every element being a linear function of t, so det(T) = 0 gives us a cubic in t. Since we know the geometrical interpretation of the equation, we know the cubic should only have one positive root.

Extrapolation

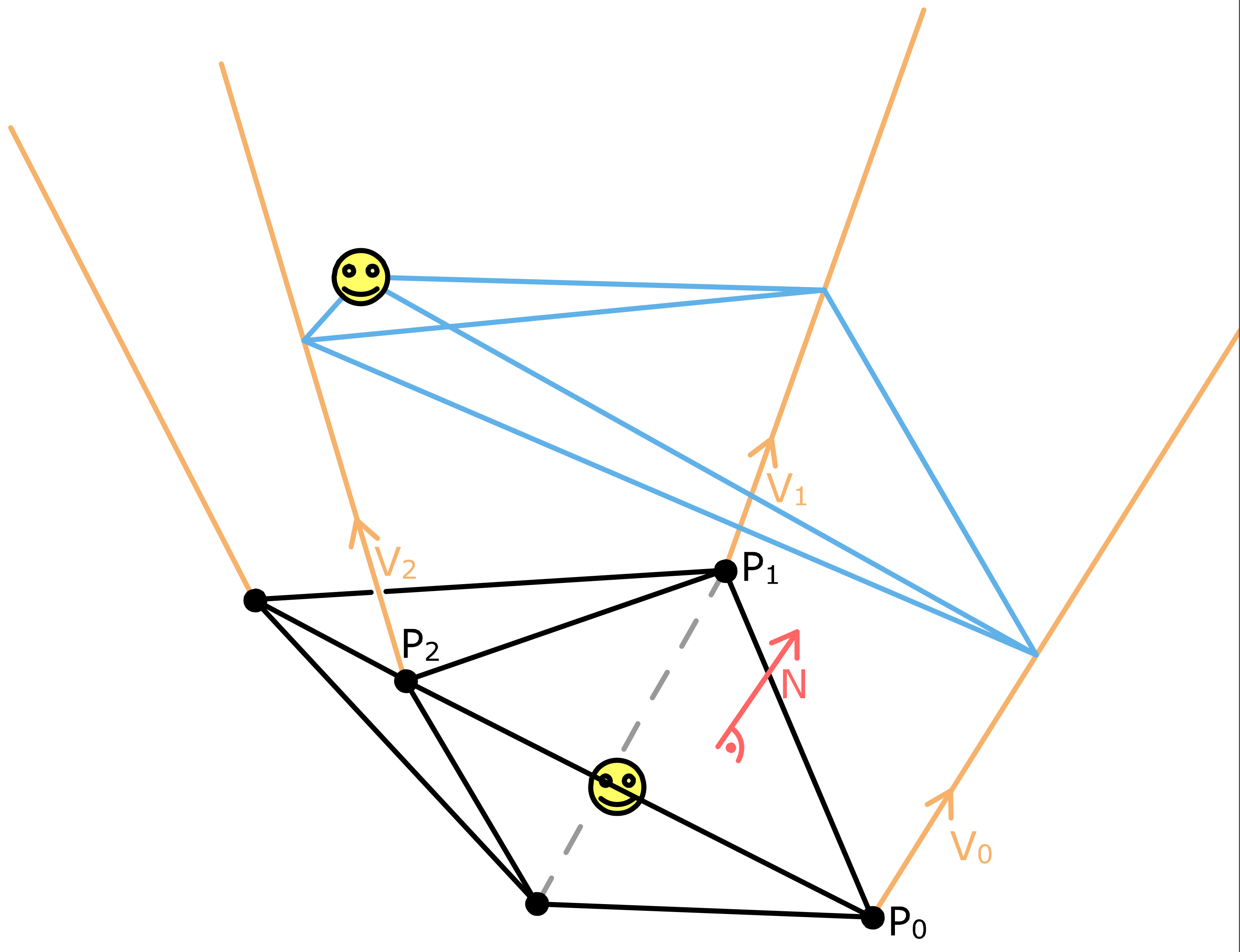
- Once we have t, plug it into the ray equations to get the triangle
- Barycentric coords from the triangle



Once we've found t, we can calculate barycentric coordinates of the object within the blue triangle and these are our weights for the 3 probes forming that outer cell.

Extrapolation - Search

- Search works the same way
  - Outer cells also have exactly 4 neighbours each
  - Test position dot normal: if negative, go to the tetrahedron below the hull face
  - Otherwise check for the most negative barycentric coord as usual



When performing the search, we pretty much proceed as before. The only difference is that we first check if we're still above the hull face by testing the dot with the (red) normal. Outer cells have 4 neighbours, just like tetrahedra.

Data

- Arrays of
  - probe positions (probe_count * 3 floats)
  - SH coefficients (probe_count * 27 floats)
  - hull rays (hull_probe_count * 3 floats)
  - tetrahedra
    - indices of the 4 vertices
    - indices of the 4 neighbours
    - matrix

That's it on the algorithmic side. We need this much data to store the structure of tetrahedra: probe positions, ray directions for the hull probes, and the index arrays for the tetrahedra. There's also the per-tetrahedron matrix which caches some things, but is not necessary.

It is of course possible to have a much more compact representation if half-precision floats are sufficient for the coefficients. 16 bits are typically sufficient for indices too.

Data

- Tetrahedron

vertex0	vertex1	vertex2	vertex3
neighbour0	neighbour1	neighbour2	neighbour3
matrix			

- An index to a neighbour is at the same position as the only vertex not shared with that neighbour
  - So e.g. vertex2 is the one not shared with neighbour2

A tetrahedron has indices to it's vertices and to it's neighbours. It's important to put the neighbours in the same order as vertices, i.e. neighbour at the position of the vertex not being shared with the current tetrahedron. This way when we find out the barycentric coordinate for vertex2 is the most negative one, we know neighbour2 is the one we need next.

Data

- Tetrahedron

vertex0	vertex1	vertex2	vertex3
neighbour0	neighbour1	neighbour2	neighbour3
matrix			

- For tetrahedra, a 3x3 matrix allows calculating barycentric coords with a matrix mult
  - Storing it saves us a matrix inverse and some more ops
  - barycentric_coords = matrix*(object_pos - vertex0_pos)

To calculate barycentric coordinates for a tetrahedron, we needed an inverse of a 3x3 matrix. The matrix only depends on the tetrahedron shape, so if we store it in the tetrahedron, we avoid calculating the inverse every time for the cost of some memory.

Data

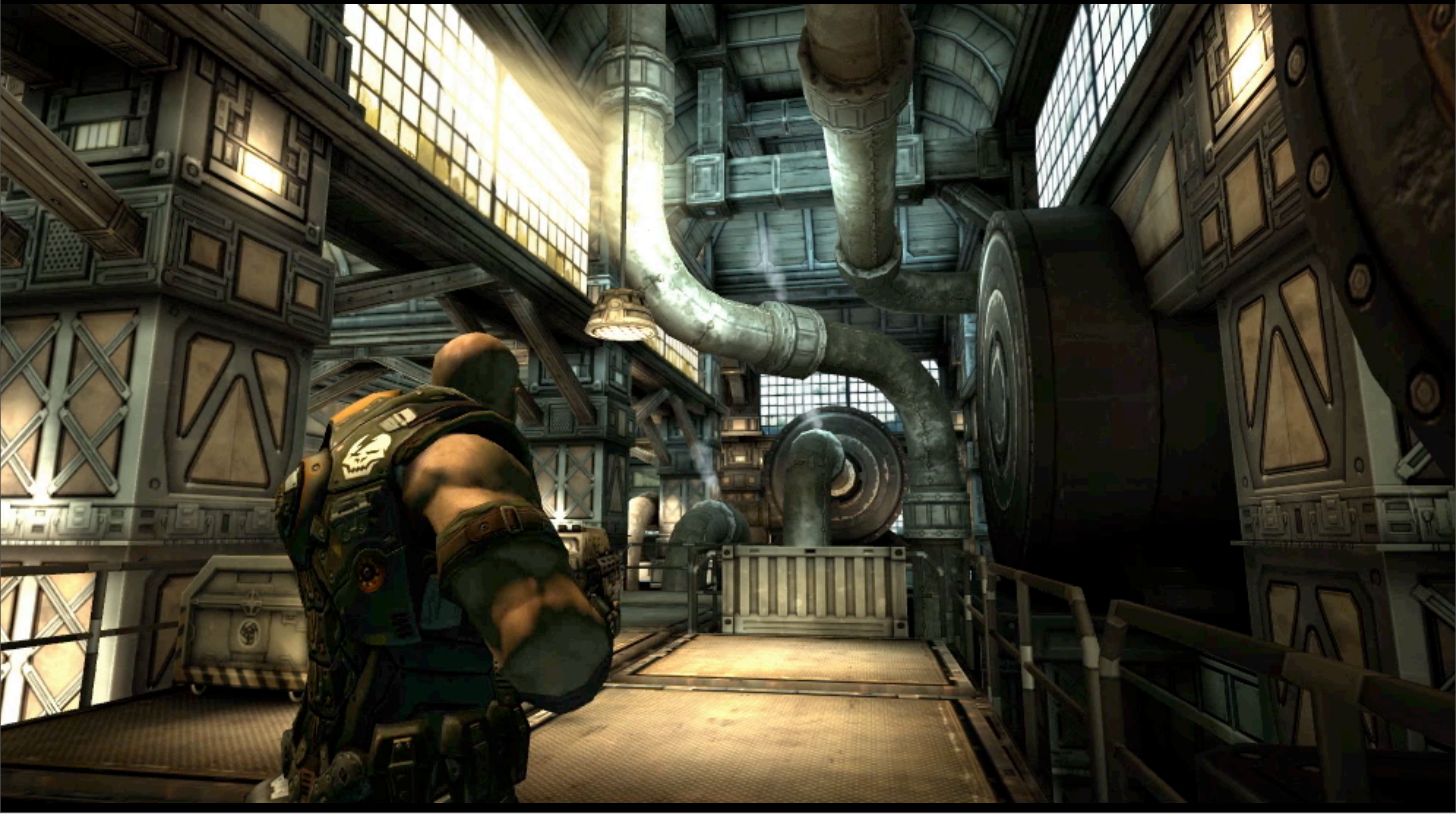
- Outer cell

vertex3 = -1, as it's an outer cell

vertex0	vertex1	vertex2	vertex3
neighbour0	neighbour1	neighbour2	neighbour3
matrix			

- For outer cells a 3x4 matrix for calculating coefficients of the cubic in monic form $t^3+pt^2+qt+r = 0$
  - [p q r] = matrix*[object_pos 1]

For outer cells the matrix caches all the calculations needed to get the coefficients of the cubic. So finding the triangle containing the object amounts to multiplying the object's position with the matrix and solving the cubic.



A level from Shadowgun to show light probes in action. The character is lit with an interpolated light probe, calculated per-vertex. That light is multiplied with a per-pixel directional light, so that the character is always lit, but also picks up the light from the various sources. The actual game runs at 60 fps on ipad2, with a bunch of enemies lit the same way, running around on the screen.

Performance

- Shadowgun: 0.5k probes, 1.5k tetrahedra: 130kB + 54kB(coeffs)
- This test: 1.5k probes, 6k tetrahedra: 510kB + 162kB(coeffs)
- 1000 queries on an iPad2
  - Typical case (hit on a first or second try): 0.5ms
  - Bad case (teleported away): 1ms
  - Worst possible case: 2.5ms

That was about one-fourth of the level. The entire level contains 500 probes.

The test level I used had 1.5k probes, which in total gave less than 700kB of data. Running 1000 queries on an iPad2 took 0.5ms if all the objects hit the typical case, i.e. found the weights right away or after one step into a neighbour. It took about twice as long when all the objects were either just instantiated far from the centre or teleported far away.

Probe placement

- For large scenes it's nice to have automatic probe placement
- It's good news that the probes can be placed anywhere
  - The placement tool can work freely
  - Automatically placed probes can be adjusted
  - More probes can be added in key areas
  - Groups of probes can be parented to prefabs

So we have an algorithm that allows us to place probes anywhere and we no longer have to (or should) create regular grids. It would be good to have automatic placement, but the good news is that the algorithm can now place probes everywhere and manual tweaking afterwards is possible as well.

Probe placement

- Adding probes based on the knowledge of the gameplay
  - Racing game? Along the racetrack, easy
  - Characters walking over a navmesh? Place over the navmesh
- Oversampling and pruning
  - Oversample the GI solution
  - Then prune the probes
    - Find a minimal set representing the original set with error below a given threshold

When automatically placing probes, it helps a lot if the algorithm understands the gameplay. In a racing game you probably only need a stripe of probes along the race track. In other games you could place them over the navmesh. You can also do some frequency analysis: flood the level with probes, bake lighting and then try to discard as many probes as possible according to some error metric.

Thanks for listening!

References

- [IrrVol] - Irradiance Volumes for Games, GDCE 2005, Natalya Tatarchuk
- [Cars2] - Rendering in Cars 2, SIGGRAPH 2011, Chris Hall, Rob Hall, Dave Edwards
- [MM] - Mega Meshes, GDC 2011, Michał Iwanicki, Ben Sugden
- [RTCD] - Real-Time Collision Detection, Christer Ericson
- [NumRob] - Numerical Robustness, GDC 2007, Christer Ericson
- [JShewchuk] - Adaptive Precision Floating-Point Arithmetic and Fast Robust Predicates for Computational Geometry, Jonathan Shewchuk cs.cmu.edu/~quake/robust.html
- [TetGen] - tetgen.berlios.de
- [PSloan] - Stupid Spherical Harmonics Tricks, Peter-Pike Sloan
- [SHL] - Spherical Harmonic Lighting: The Gritty Details, Robin Green
- [MF] - Shadowgun is an iOS and Android game by MADFINGER Games

Appendix

- Tetrahedralisation
  - Bowyer-Watson seems to be _the_ algorithm of choice
  - The method of finding the convex hull one dimension higher might be elegant and universal, but definitely not practical above 2D
  - Numerically robust implementation of the incircle and orientation tests available from [JShewchuk]
  - If you need a ready solution, [TetGen] by Hang Si is very decent and has some additional, potentially useful functionality, like tetrahedral mesh refinement

Appendix

- Random notes
  - Sampling SH in the shader
    - A nicely vectorised implementation at the end of [PSloan]
  - A Trick: once you have the interpolated probe, you can project any additional real-time lights on top of it
    - It’s approximate, but super-cheap for the CPU, completely free on the GPU
  - If the worst-case search time is unacceptable, even a very shallow BVH improves the initial guess a lot

Appendix

- Numerical precision when searching
  - The entire space is covered, but the calculations have limited precision
  - In rare cases a point might turn out to be “between” two adjacent tetrahedra
  - Fixed by checking if the next tetrahedron isn’t the one we just came from - if so, we’re at the border and we can return whichever one
  - Limit the total number of iterations
  - Read more: [NumRob] and [RTCD]

Appendix

```
inline void GetBarycentricCoordinatesForInnerTetrahedron (const dynamic_array<Vector3f>& vertices, const Vector3f& p, const Tetrahedron& tet, Vector4f& coords)
{
    Vector3f mult = tet.matrix.MultiplyVector3(p - vertices[tet.indices[3]]);
    coords.x = mult.x;
    coords.y = mult.y;
    coords.z = mult.z;
    coords.w = 1.0f - mult.x - mult.y - mult.z;
}

inline void GetBarycentricCoordinatesForOuterCell (const dynamic_array<Vector3f>& vertices, const dynamic_array<Vector3f>& hullRays, const Vector3f& p, const Tetrahedron& tet,
Vector4f& coords, float& t)
{
    const int (&ind)[4] = tet.indices;
    const Vector3f& v0 = vertices[ind[0]], v1 = vertices[ind[1]], v2 = vertices[ind[2]];
    t = Dot(p - v0, TriangleNormal(v0, v1, v2));
    if (t < 0)
    {
           // p is below the hull surface of this tetrahedron, so let's just return the 4th barycentric coordinate
           // as the lowest (and negative), so that the tetrahedron adjacent at the base gets tested next
           coords.Set(0, 0, 0, -1);
           return;
    }

    // CalculateOuterTetrahedraMatrices() prepares the Tetrahedron.matrix, so that
    // the coefficients of the cubic can be found just like that:
    Vector3f polyCoeffs = tet.matrix.MultiplyPoint3(p);
    // If the polynomial degenerated to quadratic, the unused ind[3] will be set to -2 instead of -1
    t = ind[3] == -1 ? CubicPolynomialRoot(polyCoeffs.x, polyCoeffs.y, polyCoeffs.z) : QuadraticPolynomialRoot(polyCoeffs.x, polyCoeffs.y, polyCoeffs.z);

    // We could directly calculate the barycentric coords by plugging t into a*(A + t*Ap) + b*(B + t*Bp) = C + t*Cp, checking which coord to ignore
    // and using the two other equations, but it's actually almost the same as using BarycentricCoordinates3DTriangle()
    Vector3f tri[3];
    tri[0] = v0 + hullRays[ind[0]]*t;
    tri[1] = v1 + hullRays[ind[1]]*t;
    tri[2] = v2 + hullRays[ind[2]]*t;
    BarycentricCoordinates3DTriangle(tri, p, coords);
    coords.w = 0;
}
```

A plain C++ implementation was easily fast enough for the typical probe set sizes we would get.

Appendix

```
void GetLightProbeInterpolationWeights (const LightProbeCloudData& data, const Vector3f& position, int& tetIndex, Vector4f& weights, float& t, int& steps)
{
    // If we don't have an initial guess, always start from tetrahedron 0.
    // Tetrahedron 0 is picked to be roughly in the center of the probe cloud,
    // to minimize the number of steps to any other tetrahedron.
    const int tetCount = data.tetrahedra.size();
    if (tetIndex < 0 || tetIndex >= tetCount)
            tetIndex = 0;

    steps = 0;
    for (; steps < tetCount; steps++)
    {
            // Check if we're in the current "best guess" tetrahedron
            const Tetrahedron& tet = data.tetrahedra[tetIndex];
            GetBarycentricCoordinates(data.bakedPositions, data.hullRays, position, tet, weights, t);
            if (weights.x >= 0.0f && weights.y >= 0.0f && weights.z >= 0.0f && weights.w >= 0.0f)
            {
                 // Success!
                 return;
            }

            // Otherwise find the smallest barycentric coord and move in that direction
            if (weights.x < weights.y && weights.x < weights.z && weights.x < weights.w)
                 tetIndex = tet.neighbors[0];
            else if (weights.y < weights.z && weights.y < weights.w)
                 tetIndex = tet.neighbors[1];
            else if (weights.z < weights.w)
                 tetIndex = tet.neighbors[2];
            else
                 tetIndex = tet.neighbors[3];

            // There's a chance the position lies "between" two tetrahedra, i.e. both return a slightly negative weight
            // due to numerical errors and we ping-pong between them. We could be detecting if the next tet index
            // is the one we came from. But we can also let it reach the max steps count and see if that ever happens in practice.
    }
}

inline void GetBarycentricCoordinates(const dynamic_array<Vector3f>& vertices, const dynamic_array<Vector3f>& hullRays, const Vector3f& p, const Tetrahedron& tet, Vector4f& coords, float& t)
{
    if (tet.indices[3] >= 0)
            GetBarycentricCoordinatesForInnerTetrahedron (vertices, p, tet, coords);
    else
            GetBarycentricCoordinatesForOuterCell (vertices, hullRays, p, tet, coords, t);
}
```

The for loop typically spins here only once or twice.

Appendix

```
void CalculateOuterCellsMatrices(LightProbeCloudData& data, int innerTetrahedraCount)
{
    for (int i = innerTetrahedraCount; i < data.tetrahedra.size(); i++)
    {
        Vector3f V[3];
        for (int j = 0; j < 3; j++)
            V[j] = data.hullRays[data.tetrahedra[i].indices[j]];

        Vector3f P[3];
        for (int j = 0; j < 3; j++)
            P[j] = data.vertices[data.tetrahedra[i].indices[j]];

        Vector3f A = P[0] - P[2];
        Vector3f Ap = V[0] - V[2];
        Vector3f B = P[1] - P[2];
        Vector3f Bp = V[1] - V[2];
        //Vector3f C = p - P[2];
        Vector3f P2 = P[2];
        Vector3f Cp = -V[2];

        Matrix3x4f& m = data.tetrahedra[i].matrix;

        // output.x =
        // input.x*
        m[0] = Ap.y*Bp.z
                -Ap.z*Bp.y;
        // input.y*
        m[3] = -Ap.x*Bp.z
                +Ap.z*Bp.x;
        // input.z*
        m[6] = +Ap.x*Bp.y
                -Ap.y*Bp.x;
        // 1*
        m[9] = +A.x*Bp.y*Cp.z
                -A.y*Bp.x*Cp.z
                +Ap.x*B.y*Cp.z
                -Ap.y*B.x*Cp.z
                +A.z*Bp.x*Cp.y
                -A.z*Bp.y*Cp.x
                +Ap.z*B.x*Cp.y
                -Ap.z*B.y*Cp.x
                -A.x*Bp.z*Cp.y
                +A.y*Bp.z*Cp.x
                -Ap.x*B.z*Cp.y
                +Ap.y*B.z*Cp.x;

        m[9] -= P2.x*m[0] + P2.y*m[3] + P2.z*m[6];

        // output.y =
        // input.x*
        m[1] = +Ap.y*B.z
                +A.y*Bp.z
                -Ap.z*B.y
                -A.z*Bp.y;
        // input.y*
        m[4] = -A.x*Bp.z
                -Ap.x*B.z
                +A.z*Bp.x
                +Ap.z*B.x;
        // input.z*
        m[7] = +A.x*Bp.y
                -A.y*Bp.x
                +Ap.x*B.y
                -Ap.y*B.x;
        // 1*
        m[10] =         +A.x*B.y*Cp.z
                -A.y*B.x*Cp.z
                -A.x*B.z*Cp.y
                +A.y*B.z*Cp.x
                +A.z*B.x*Cp.y
                -A.z*B.y*Cp.x;

        m[10] -= P2.x*m[1] + P2.y*m[4] + P2.z*m[7];

        // output.z =
        // input.x*
        m[2] = -A.z*B.y
                +A.y*B.z;
        // input.y*
        m[5] = -A.x*B.z
                +A.z*B.x;
        // input.z*
        m[8] = +A.x*B.y
                -A.y*B.x;
        // 1*
        m[11] = 0.0f;

        m[11] -= P2.x*m[2] + P2.y*m[5] + P2.z*m[8];

        float a =
            Ap.x*Bp.y*Cp.z
            -Ap.y*Bp.x*Cp.z
            +Ap.z*Bp.x*Cp.y
            -Ap.z*Bp.y*Cp.x
            +Ap.y*Bp.z*Cp.x
            -Ap.x*Bp.z*Cp.y;

        if (Abs(a) > EPSILON)
        {
            // d is not zero, so the polymial at^3 + bt^2 + ct + d = 0 is actually cubic
            // and we can simplify to the monic form t^3 + pt^2 + qt + r = 0
            for (int k = 0; k < 12; k++)
                m[k] /= a;
        }
        else
        {
            // It's actually a quadratic or even linear equation,
            // Set the last vertex index of the outer cell to -2
            // instead of -1, so at runtime we know the equation is
            // pt^2 + qt + r = 0 and not t^3 + pt^2 + qt + r = 0
            data.tetrahedra[i].indices[3] = -2;
        }
    }
}
```

The data for the tetrahedra and outer cells is filled out at bake time. For outer cells the matrix is derived from det(T) = 0 and allows for calculating the coefficients of the cubic with a single matrix multiplication at runtime.