The 3D Quickhull Algorithm

Dirk Gregorius – Valve Software

Good afternoon! My name is Dirk and I am a software engineer at Valve.
This year I am going to talk about convex hull creation for collision detection.

Ask audience:
- Who attended my talk last year?
- Who is using SAT now?

- If not, who tried it? What were the problems?
- If you are interested, please step forward after the tutorial and we can discuss some of those issues!

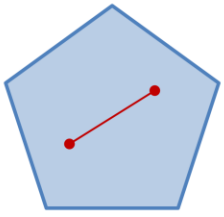
Outline

- **Introduction**
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- Implementation

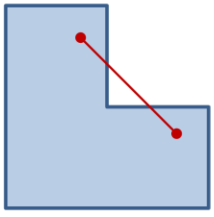
Before we start I like to quickly outline the talk.

- After a short introduction I will first start with Quickhull in 2D
- Then we continue with geometrical invariants which we need to maintain while constructing the hull to avoid numerical problems
- After the 2D introduction we will directly dive into the 3D version of Quickhull
- We then investigate geometrical and topological invariants while constructing the hull in 3D and close with some implementation details

Convexity



Convex

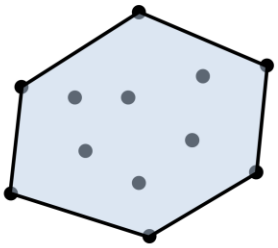


Concave

Before we start let's figure out what a convex hull is and look at an example:

- We call a shape convex, if for any two points that are *inside* the shape, the line between these two points is also inside the shape
- A concave shape does NOT satisfy this requirement as you can see in the figure on the right hand side
- If you look at concave case you see that it is easy to find two points inside the shape where the line has to leave and then enter the shape again.

Convex Hulls



A 2D set of points and its convex hull

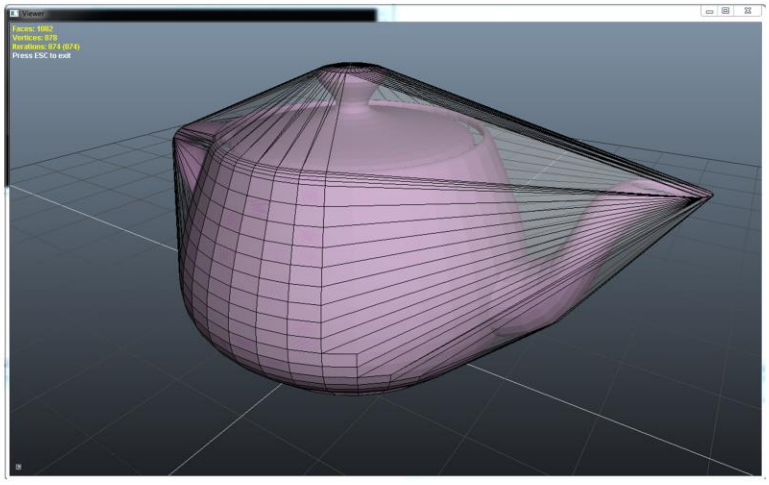
What is a convex hull?

Give a set of N input points we can now define what we are going to call a convex hull:

- Formally: A convex hull is the smallest convex set containing all input points
- Informally: If your points would by nails sticking in some piece of wood, the convex hull would be a rubber band wrapped around the outside nails.

-> This means in 2D the hull is a convex polygon defined by vertices and edges!

Convex Hulls (2)



I also like to show a 3D convex hull of a well known object – Utah Teapot.

- It is incredible how versatile this model is!
- In 2D we used the rubber band analogy to get some intuition for a convex hull
- In 3D you can think of shrink wrapping the object

-> This means in 3D the hull is a convex polyhedron defined by vertices, edges and *polygonal* faces! Note that we are not going to restrict ourselves to triangles faces only!

Convex Hulls for Collision Detection

- Dynamic game objects are usually approximated by simple shapes for collision detection
- Convex hulls are a good candidate for this
- Collision detection for convex polyhedra is well defined and robust (e.g. GJK and SAT)

Why should we use convex hulls for collision detection in games?

- Dynamic game objects are usually approximated by simpler shapes for collision detection since using the render geometry would NOT be efficient
- Convex hulls are a good candidate since they can approximate even complex geometry quite well
- Also, collision detection for convex polyhedra is well defined and robust. Think of GJK and SAT which we discussed already here in earlier tutorials

Convex Hulls in the Game

Before we start I like to show two videos to give you some idea how convex hulls are used in games.

- Show a movie of convex hulls in the game
- Show how the physics engine sees the game

Sergiy will show you how to implement awesome physics visualization right after this talk!

Quickhull

- Published by C. Barber and D. Dobkin in 1995
- Iterative algorithm that adds one point at a time
- Addresses issues with ill-defined input sets when computing the convex hull
- The output is a set of 'fat' faces that encloses all possible exact convex hulls for the input set

I hope the videos gave you an idea about the problem we are trying to solve here today.

When I started looking in convex hulls I quickly came across an algorithm called Quickhull:

- Quickhull was published by Barber and Dobkin in 1995
- It is an iterative algorithm that adds individual points one after the other to intermediate hulls.
- When implementing an algorithm like Quickhull using floating point arithmetic you cannot assume that your computations will be exact which becomes a problem when your input set is ill-defined (e.g. nearly identical points)
- Quickhull uses fat faces to deal with numerical issues and the output is a set of 'fat' faces that contain all possible exact convex hulls for the input set
- In the remainder of the talk will try to explain what this means in detail!

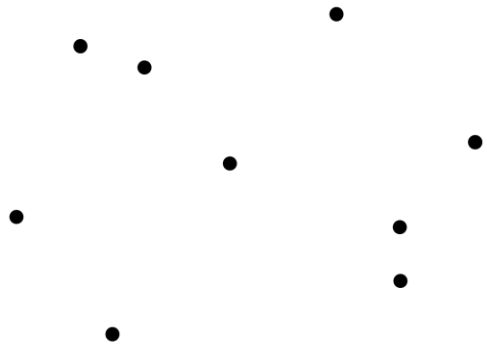
Outline

- Introduction
- **Introduce Quickhull in 2D**
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- Implementation

In this presentation I like to share what I learned about convex hulls in general and about Quickhull in particular:

- I will start outlining the algorithm in 2D first
- I like to mention, that this is not the 'real' **2D** Quickhull algorithm (which actually exists). You should think of it as an introduction of the 3D version we will investigate later in the talk
- Personally I find it often helpful to think about things in 2D first, to get a good understanding of the problem and to familiarize myself with the basic ideas

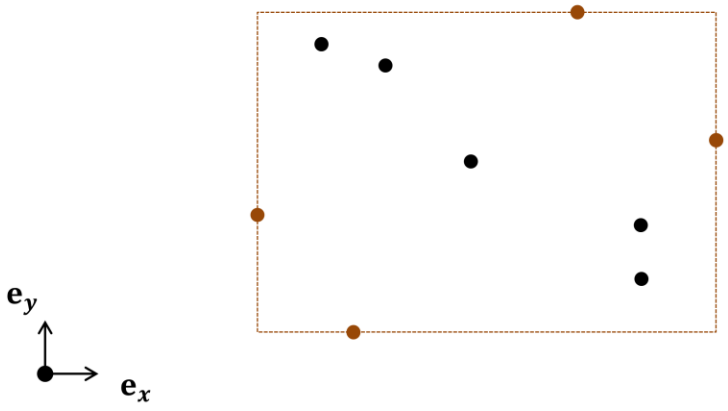
Quickhull 2D: Initial Hull



Assume we are a given set of points and we asked to build the convex hull using the Quickhull algorithm:

- The first thing we need to do in Quickhull is to build an initial hull from where we can start adding points iteratively

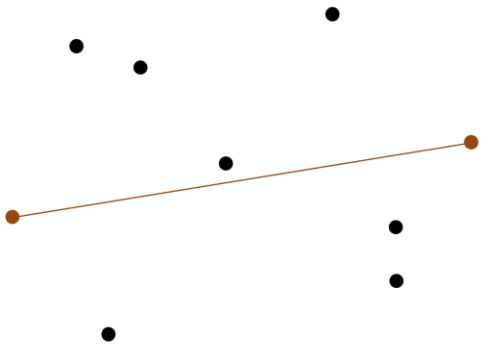
Quickhull 2D: Initial Hull (2)



To find this initial hull we start by identifying the extreme points along each cardinal axis

- This simply means we find the points with the smallest and largest x and y values.

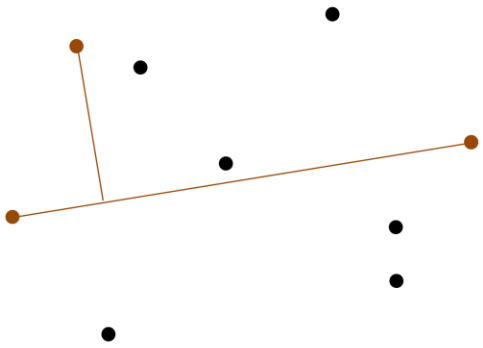
Quickhull 2D: Initial Hull (3)



From these four points we choose the pair which is furthest apart

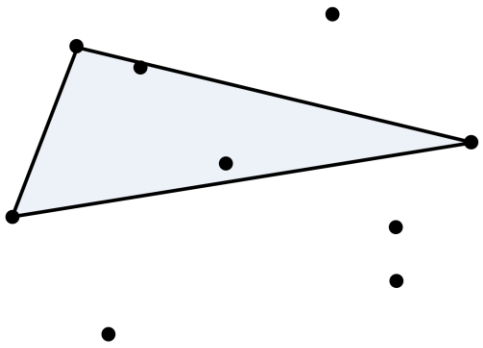
- In this example this would be the left and right-most points

Quickhull 2D: Initial Hull (4)



- Finally we search for the furthest point from the line through these two extreme points

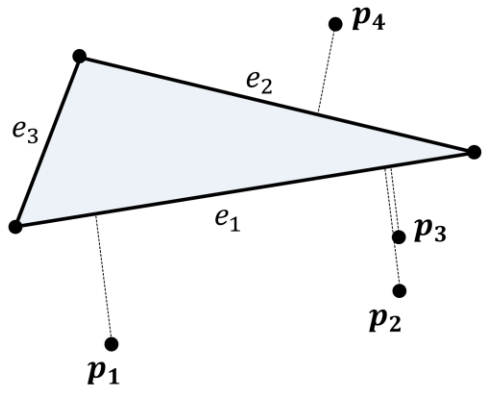
Quickhull 2D: Initial Hull (5)



These three points build our initial hull

- In 2D the initial hull is simply a triangle

Quickhull 2D: Partition Points



$CL(e_1) = \{ \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3} \}$

$CL(e_2) = \{ \boldsymbol{p_4} \}$

$CL(e_3) = \{ \ \}$

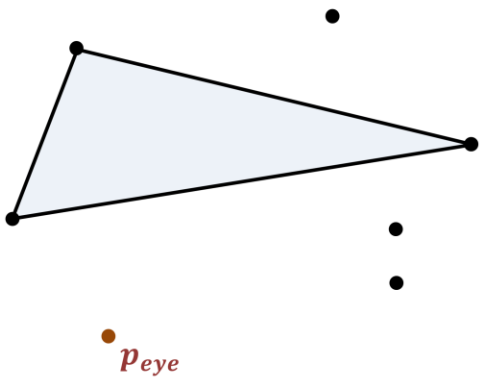
Before we start adding new points to the initial hull we have to do some bookkeeping work:

- The next step is to partition the remaining points and assign each point to its closest face
- We can also remove internal points since those cannot be on the final hull

What that means is that each face maintains a list of points which are outside the face plane. We call those 'conflict lists' since the points can "see" the face and therefore are potentially on the final hull. This is a clever way of managing the vertices since we don't need to iterate all vertices when adding a new vertex to the hull. This makes Quickhull typically O(n log n) in both 2 and 3 dimensions!

NOTE:
Please don't get confused here. Since I am presenting in 2D and 3D and some terminology overlaps I will use the terms 'Edge' and 'Face' interchangeable! This will usually help when we go to 3D later in the talk!

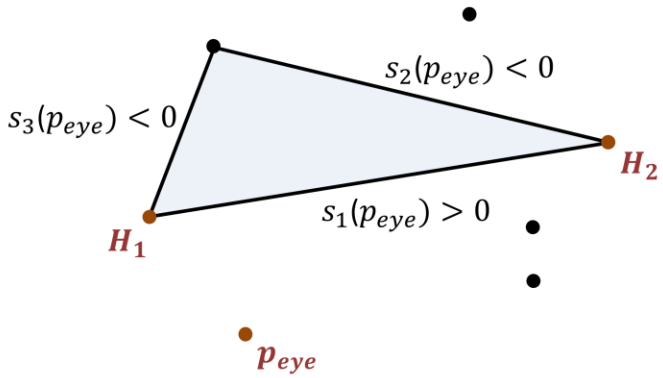
Quickhull 2D: Find Next Vertex



The next step is to add a new point to our intermediate hull. We iterate our conflict lists and find the point p with the largest distance from the hull:

- Let's call this point the **eye** point
- Adding this new point requires several sub steps

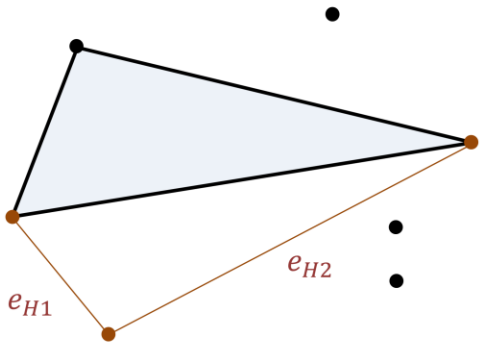
Quickhull 2D: Find Horizon



First we need to identify all faces that are visible from the newly added point since these faces cannot be on the hull:

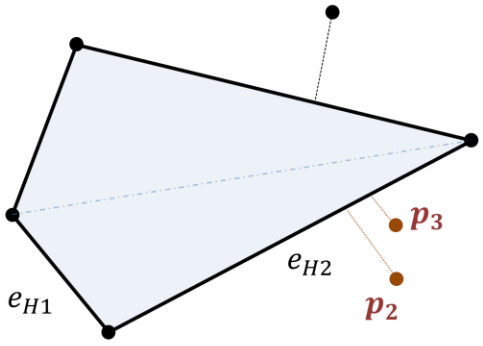
- A face is visible if the new point is in front of the face plane.
- We can use simple plane tests to classify the new point against each face!
- The next step is then two find the two vertices that connect a visible with a non-visible face.
- We call these two vertices the horizon

Quickhull 2D: Add Point to Hull



- Once we identified the two horizon vertices we then create two new faces for each horizon vertex to connect the new vertex to hull

Quickhull 2D: Partition Orphans



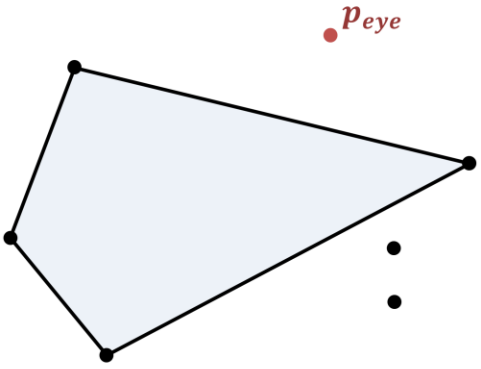
$CL(e_{H1}) = \{\}$

$CL(e_{H2}) = \{p_2, p_3\}$

After building the new faces some old faces became obsolete

- Before we can delete these faces we need to handle their conflict lists since these conflict points can still be on the final hull
- We handle this by simply partitioning these orphaned vertices to the new faces

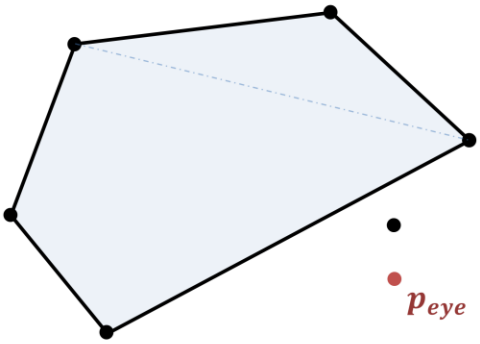
Quickhull 2D: Add Point to Hull



Finally we can now remove all old faces which were visible from the new point and therefore cannot be on the hull anymore

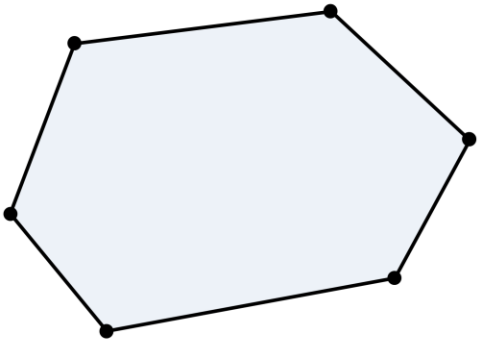
- This closes the iteration and we repeat those step until all conflict list are empty
- In our example here we continue and grab the next eye point and add it to hull as we just learned

Quickhull 2D: Repeat (1)



And we do it one more time to find our final hull…

Quickhull 2D: Repeat (2)



When there are no more vertices (which means all conflict lists are empty) we are done!

Quickhull 2D: Summary

- As you can see the basic Quickhull algorithm is pretty simple
- In 3D one major implementation difficulty arises from managing the lists of vertices, edges, faces and conflicts
- The other difficulty is dealing with numerical imprecision in our plane tests.

- As you can see the basic ideas should be pretty easy to understand
- In 3D the major implementation difficulties actually arise from managing the lists of vertices, edges, faces and conflicts
- The other difficulty is dealing with numerical imprecision when classifying points using plane tests.

Outline

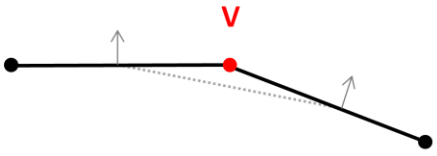
- Introduction
- Introduce Quickhull in 2D
- **Quickhull 2D Invariants**
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- Implementation

So far we pretended that our mathematical operations are exact.

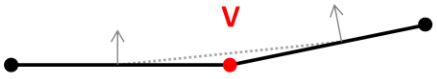
- Of course this is not true in floating point arithmetic.
- Let's investigate how we can deal with those problems.

Quickhull 2D: Invariants

Vertex **v** is convex



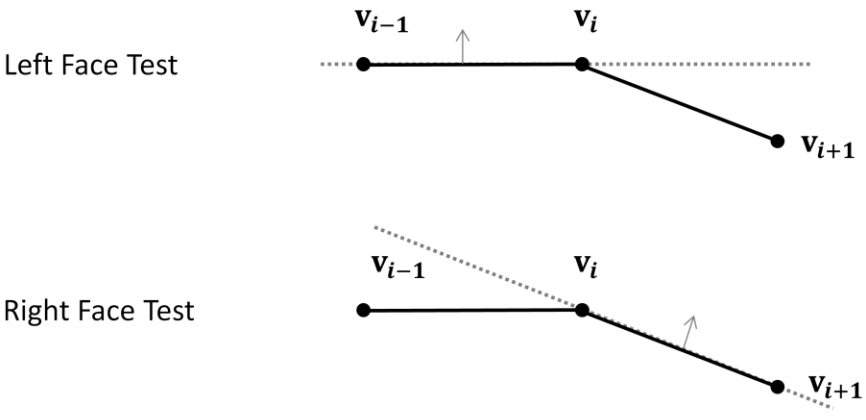
Vertex **v** is concave



When building a convex hull we must maintain geometric invariants while constructing the hull:

- In 2D we must guarantee that every vertex is convex
- This should be obvious since otherwise it would be simple to find a line between two points inside the hull that would leave and enter as shown in the slide at the beginning of the talk
- I tried to hint this with the dotted line between the normals in the slide

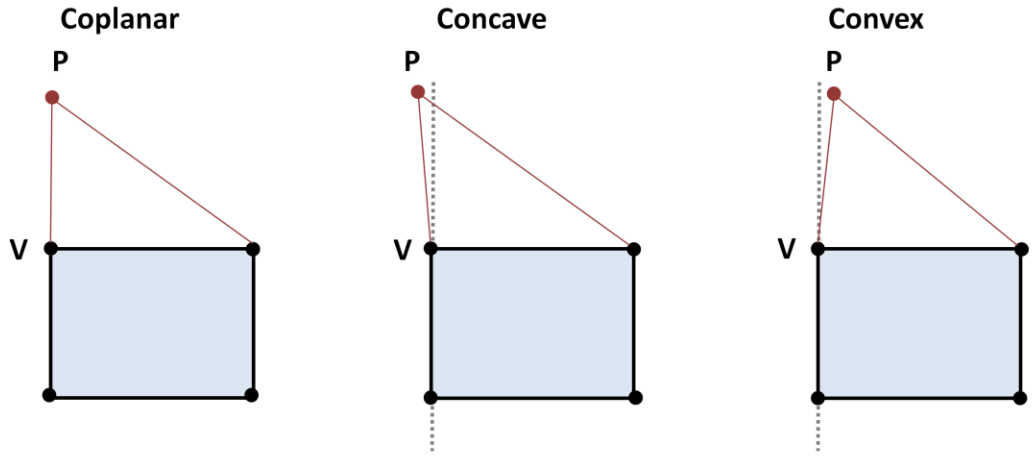
Quickhull 2D: Convexity Test



Next we need to define what a **convex** vertex is and how we can test this vertex for convexity:

For each vertex:
- First test if right vertex is below the left face
- Then test if left vertex if it is below the right face
- If both tests are true the **vertex** is convex, otherwise it must be concave or coplanar

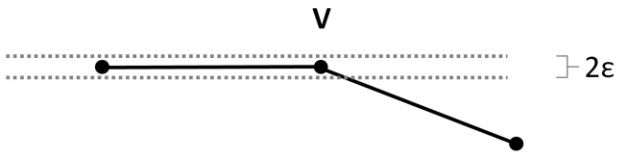
Quickhull 2D: Numerical Robustness



Let's now look at an example where non-convex vertices might become an issue:

- Whenever we add a point which is collinear with an existing face things can become fuzzy
- A small variation of point **P** will define whether the vertex **V** will remain on the hull or not
- Ideally we would like to have more stability such that for very *small* variations within some tolerance we would get the same result
- Note that the point **P** is not actually moving, but can end up on either side of the plane *just* due to numerical imprecision

Quickhull 2D: Fat Planes



$$P := \mathbf{n} \cdot \boldsymbol{x} - \mathrm{d}$$

$$\mathrm{s} = \mathbf{n} \cdot \mathbf{v} - \mathrm{d} \Rightarrow \begin{cases} s > \varepsilon, & (in\ front\ of\ plane) \\ s < -\varepsilon, & (behind\ plane) \\ -\varepsilon \leq s \leq \varepsilon, & (on\ plane) \end{cases}$$

A common approach to deal with these kinds of numerical problems is to use so called *fat* planes

- Instead of comparing directly against zero we now compare against some epsilon value

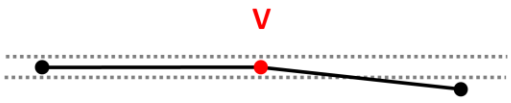
We still can classify points when using fat planes as before:

- A point is in front of the plane if its distance is larger than epsilon
- A point is behind the plane if the distance is less than negative epsilon
- Otherwise the point must on the plane.

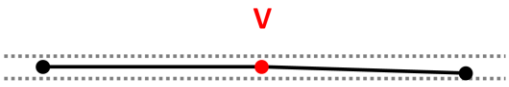
We can now define a vertex to be convex if its distance is larger then epsilon. All other points are either concave or coplanar and should be handled specially!

Quickhull 2D: Fat Planes (2)

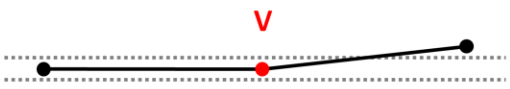
Vertex **v** is convex



Vertex **v** is coplanar



Vertex **v** is concave

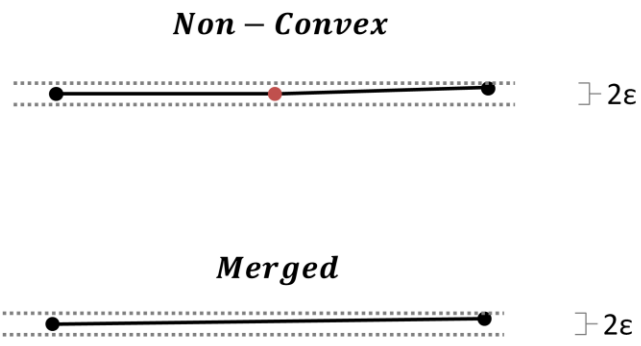


Let's have a quick look how the fat planes affect our definition of convex, coplanar and concave vertices.

For each vertex:

- Find the distance of the right vertex to the left face
- Find the distance of the left vertex to the right face
- If both distances are larger epsilon the vertex is **convex**, otherwise it must be **concave** or **coplanar**

Quickhull 2D: Face Merging



So what do we do when we encounter a non-convex vertex?

- In order to deal with non-convex vertices we can simply merge the left and right face across the vertex into a new face.
- This will correct the geometrical defect.
- In 2D we only need to inspect the two new horizon vertices at each iteration and correct those if necessary!

Quickhull 2D: Epsilon

$$\varepsilon = 2\left(\max_i\left((abs(x_i)\right) + \max_i\left((abs(y_i)\right)\right) \cdot FLT\_EPSILON$$

The final question is what epsilon we should choose for our fat planes:

- The CRT defines a static floating point epsilon but this does NOT take our input set into account.
- We like to define a relative tolerance which takes the size of the input into account
- Note that it will not be sufficient to just use the extent of the AABB since our input vertices might be just way off the origin
- So one possible solution it to choose an epsilon relative to the sum of maximum absolute coordinates of the input set

Outline

- Introduction
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- **Introduce Quickhull in 3D**
- Quickhull 3D Invariants
- Implementation

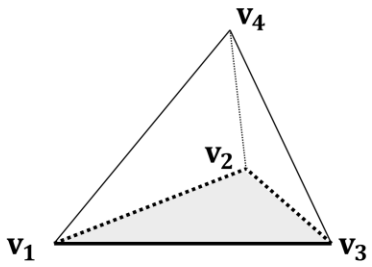
This closes the introduction of Quickhull in 2D. I hope you now have some first idea how the algorithm operates.

- In 2D there already exist good algorithms to build convex hulls which are easy and straight forward to implement.
- So if your game is 2D I recommend using one of those.

In the remainder of the talk we will discuss how we build convex hulls in 3D

- Quickhull in 3D is very similar to the version I just showed you in 2D
- The most notable difference is the construction of the horizon and we have to deal with numerical imprecision more carefully

Quickhull 3D: Initial Hull



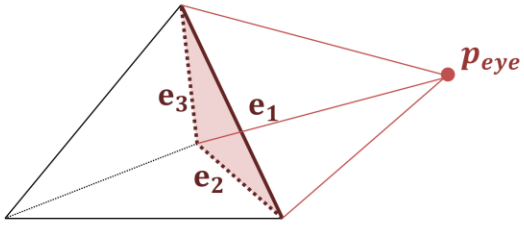
As in 2D we need to build an initial hull

- We first find the initial triangle (v1, v2, v3) as we did in 2D before
- Then we also add the furthest point from the triangle plane (here v4)
- In 3D the initial hull is now a tetrahedron
- After we build the initial hull we partition the remaining points as before into the conflict lists

Quickhull 3D: Horizon

Horizon:

H = { $\mathbf{e_1}$, $\mathbf{e_2}$, $\mathbf{e_3}$ }



We then start iteratively adding new points to the hull and grab the point with the largest distance from our conflict lists:

- This gives us the next eye point
- As in 2D we need to find the horizon again
- In 3D the horizon is a list of edges that connect visible with non-visible faces.
- You can think of it as the boundary between the visible
- Finding the horizon is a bit more involved in 3D and we will look at it in more detail in just a second

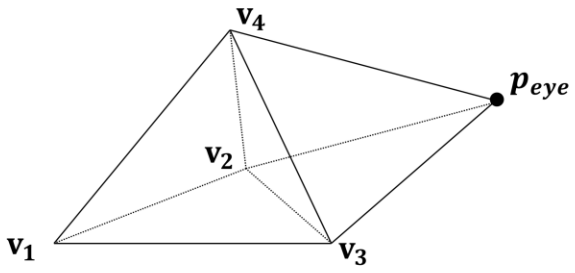
Quickhull 3D: Add point to hull

New Faces:

$F_1 = \{ \boldsymbol{p_{eye}}, \mathbf{v_4}, \mathbf{v_3} \}$

$F_2 = \{ \boldsymbol{p_{eye}}, \mathbf{v_3}, \mathbf{v_2} \}$

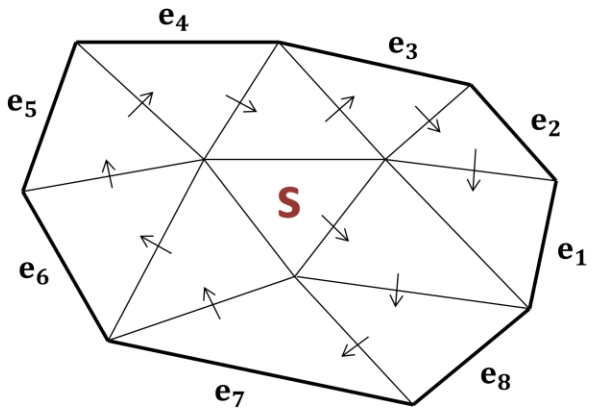
$F_3 = \{ \boldsymbol{p_{eye}}, \mathbf{v_2}, \mathbf{v_4} \}$



We then proceed with the iteration and create a new face for each horizon edge with the new eye-point

- This essentially connects the new vertex to the current hull
- Finally we partition the orphaned vertices to the new faces (F1 – F3)

Quickhull 3D: Finding the Horizon

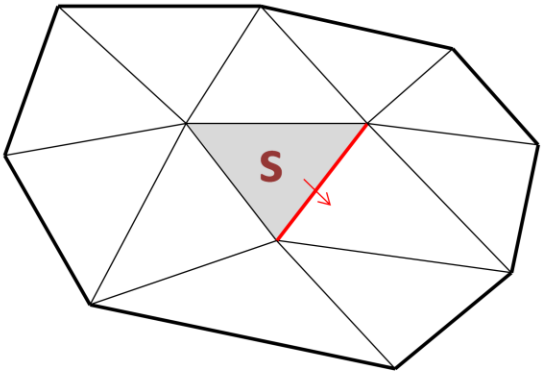


Finding the horizon in 3D is not as easy as finding two vertices as it was in 2D.

- For finding the horizon we essentially perform a flood fill starting from the conflict face **S**
- At each step we cross one edge and visit a neighboring face
- If the face is visible, we cross another edge until we find a face that is not visible from the current eye point
- We store the offending edge as part of the horizon and continue the search in the previous face
- On termination we have a list of all edges defining the horizon in a closed CCW loop
- On the slide we start at the face labeled with S and follow the arrows
- Since this is an essential operation of the hull construction let's look at this in more detail

Explain arrows in slide!

Quickhull 3D: Find Horizon

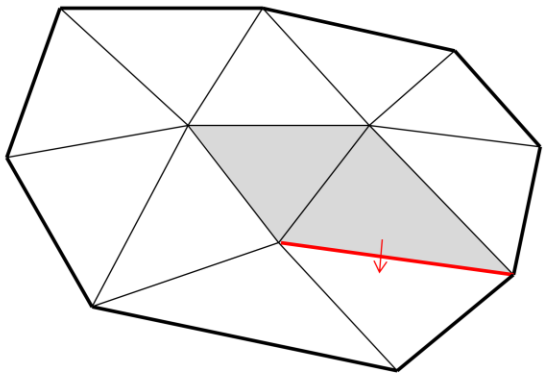


I prepared a small animation which hopefully will help to understand the horizon construction:

- I will just run it first so you get the idea and sometime a good picture says more than 1000 words
- We will then rewind and I will point out some important points

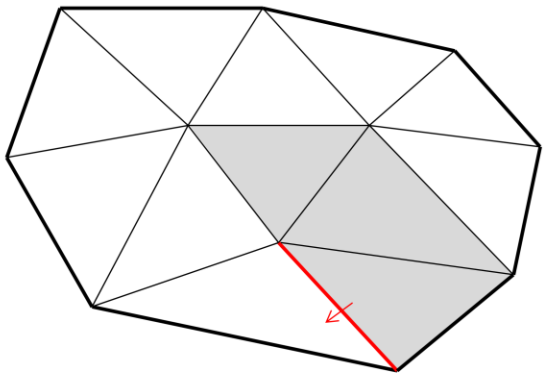
We start at the conflict face and cross the first edge

Quickhull 3D: Find Horizon



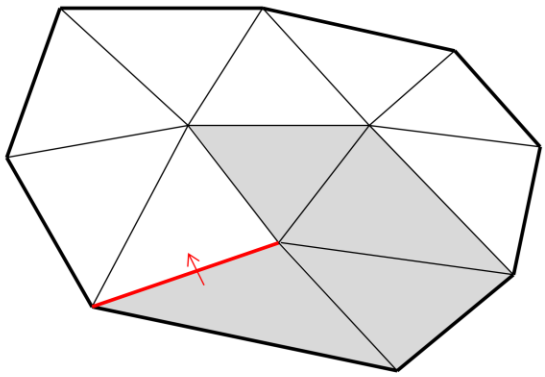
- We test the visibility of the next face (and we assume here it is visible from the eye point)
- Since it is visible we continue our search and cross the next edge

Quickhull 3D: Find Horizon



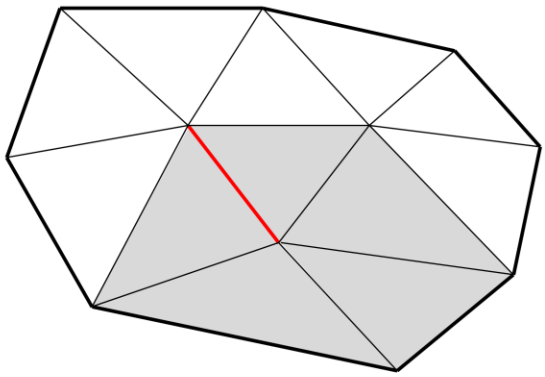
- We test the next face and since it is visible so continue and cross the next edge

Quickhull 3D: Find Horizon



- We continue these tests until we cross an edge to an invisible face

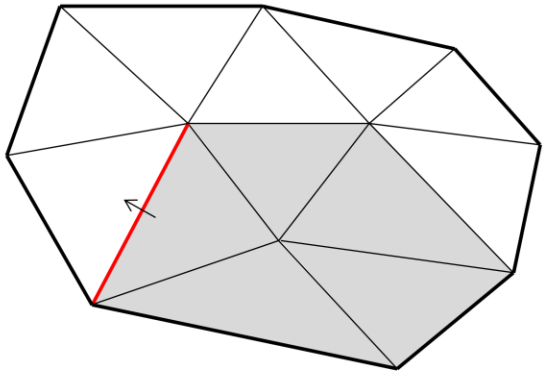
Quickhull 3D: Find Horizon



We would now cross an edge to a face we have already visited

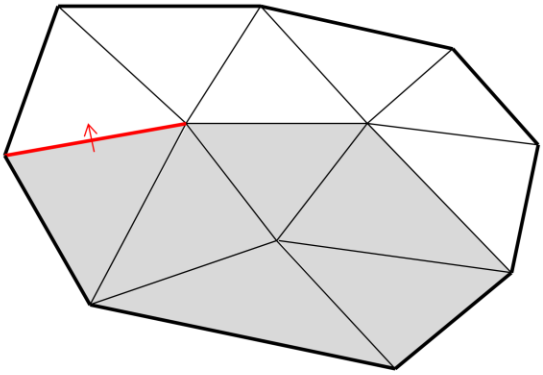
- Whenever we visit a face we will mark them as processed
- This allows to simply test if a face was already processed and can be skipped

Quickhull 3D: Find Horizon

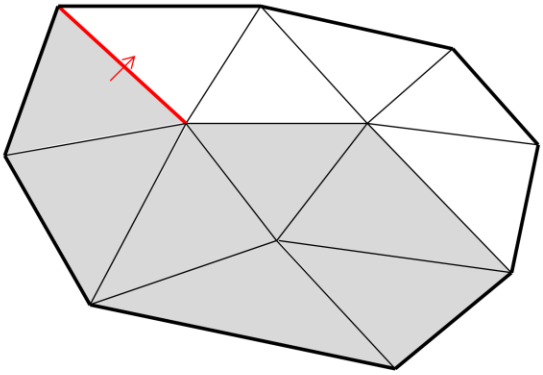


…

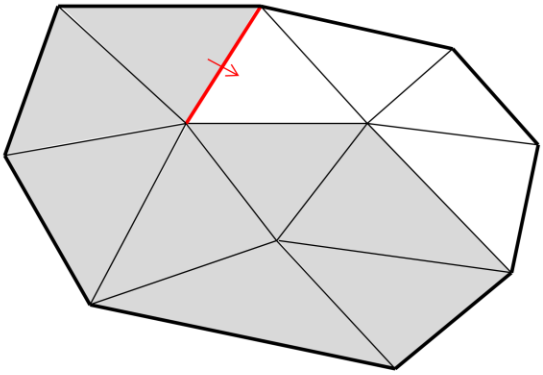
Quickhull 3D: Find Horizon



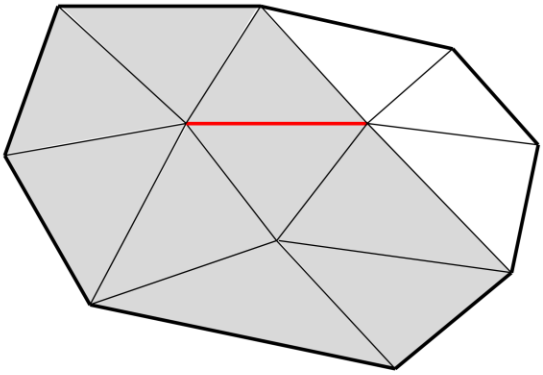
Quickhull 3D: Find Horizon



Quickhull 3D: Find Horizon

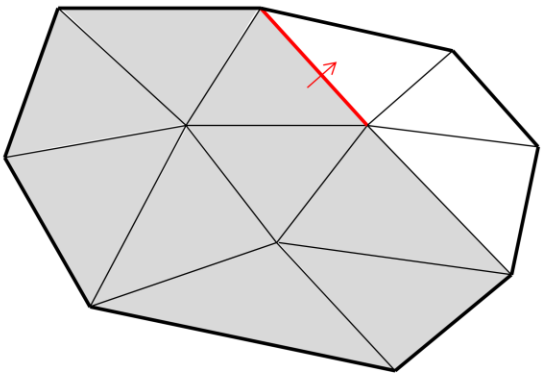


Quickhull 3D: Find Horizon

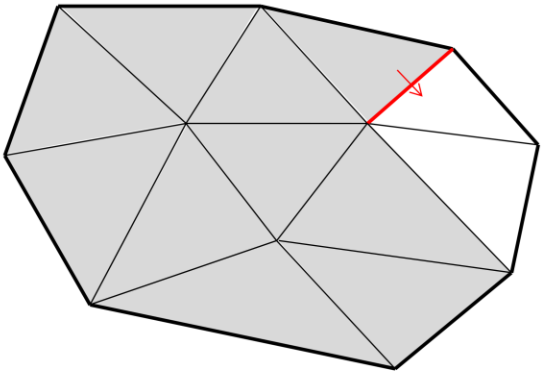


Again we already visited this face so we don’t cross this edge as well

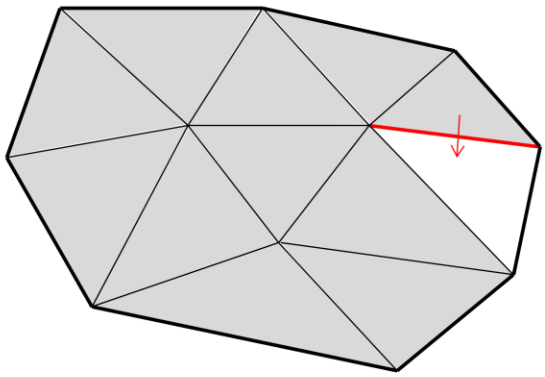
Quickhull 3D: Find Horizon



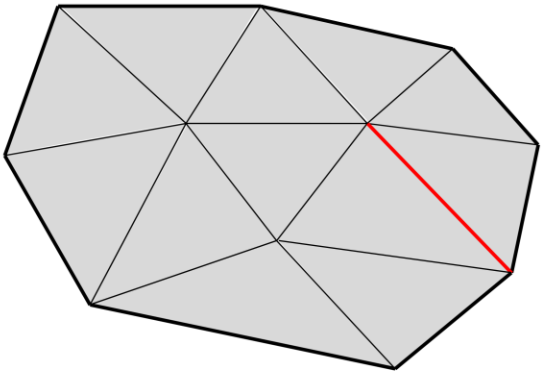
Quickhull 3D: Find Horizon



Quickhull 3D: Find Horizon

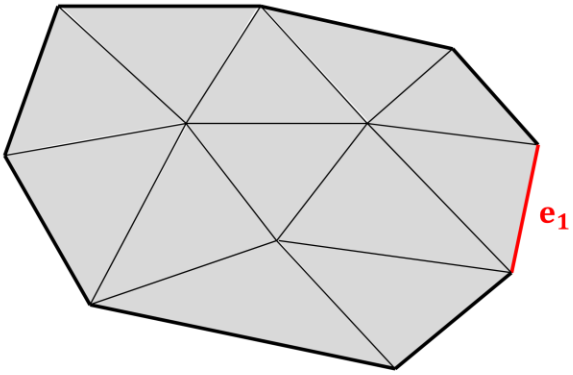


Quickhull 3D: Find Horizon



The next face was already visited so no need to cross here as well

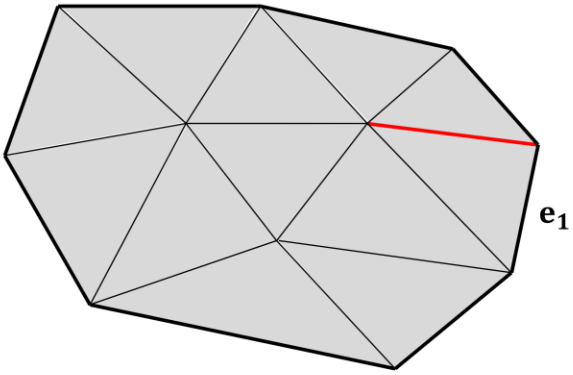
Quickhull 3D: Find Horizon



Finally we would cross the first edge that connects a visible and an invisible face

- We add the edge to the horizon list and return to the previous face

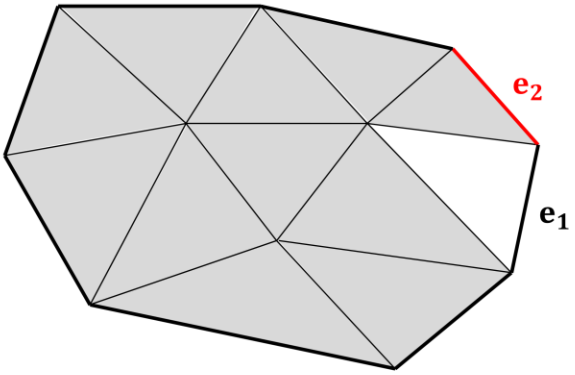
Quickhull 3D: Find Horizon



We continue with the next edge.

- Since this is the edge we crossed over to the current face we return to the predecessor of the current face

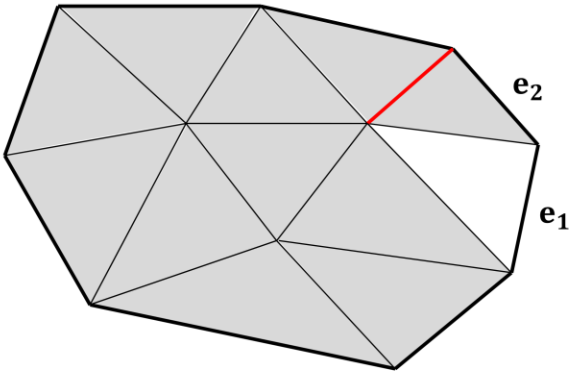
Quickhull 3D: Find Horizon



Again we cross an edge that connects a visible and invisible face

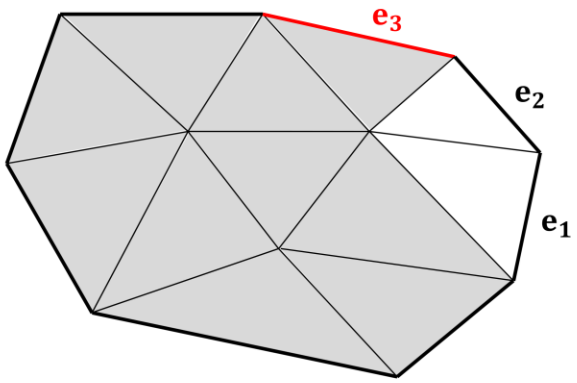
- We save that edge and add it to our horizon and return to the previous face

Quickhull 3D: Find Horizon

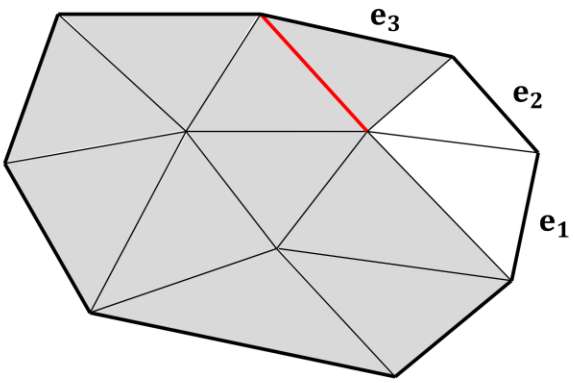


The procedure continues and collects the horizon edges until we made our way back to the start face

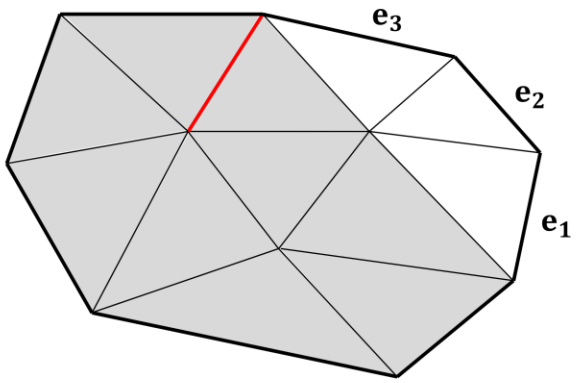
Quickhull 3D: Find Horizon



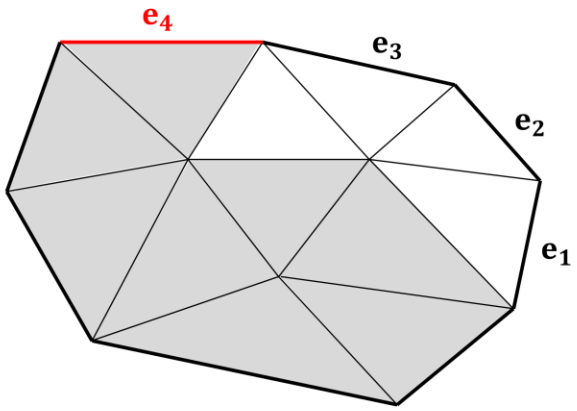
Quickhull 3D: Find Horizon



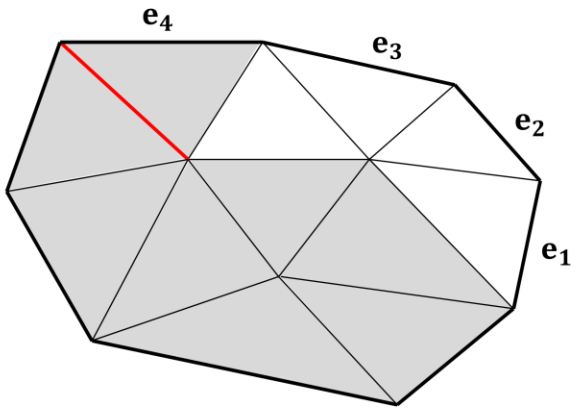
Quickhull 3D: Find Horizon



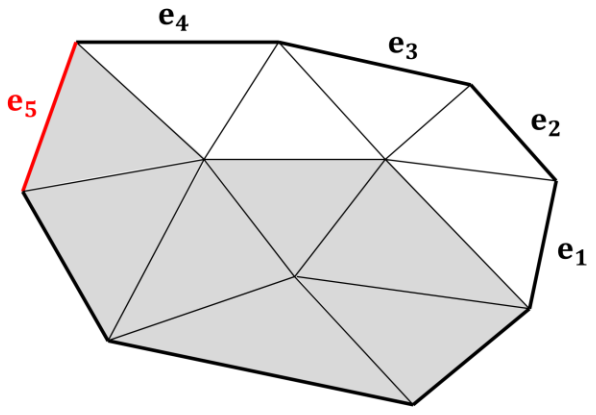
Quickhull 3D: Find Horizon



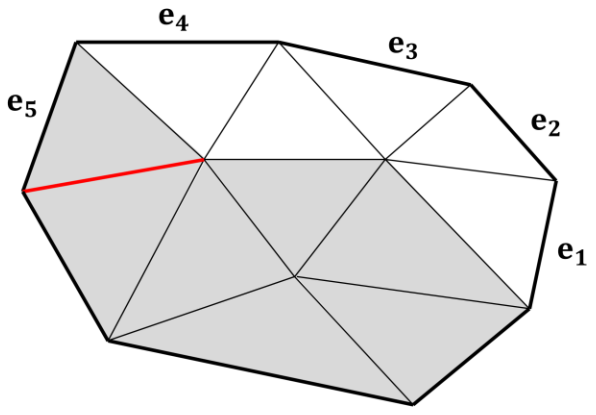
Quickhull 3D: Find Horizon



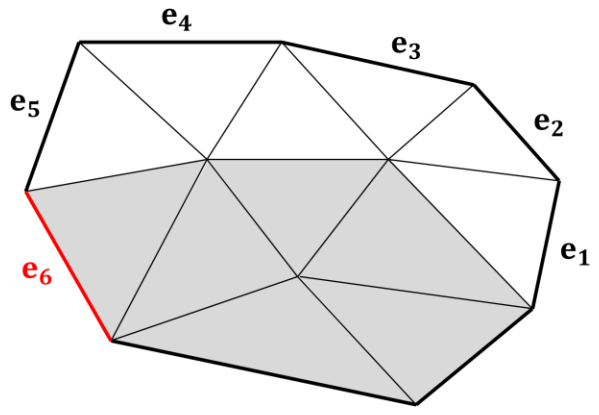
Quickhull 3D: Find Horizon



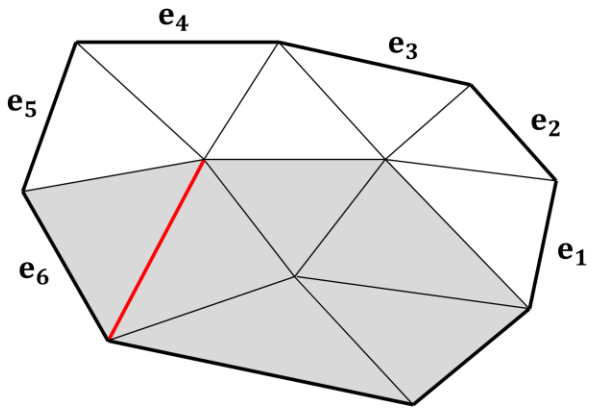
Quickhull 3D: Find Horizon



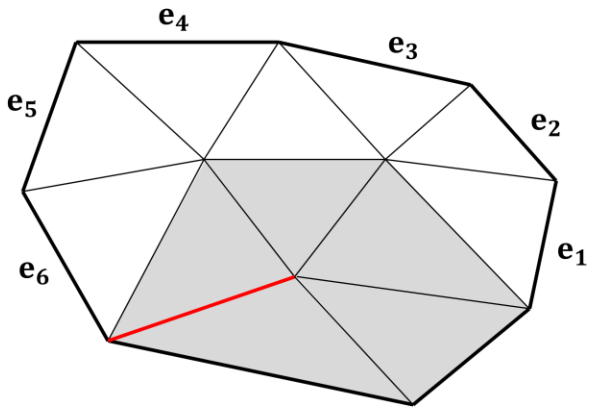
Quickhull 3D: Find Horizon



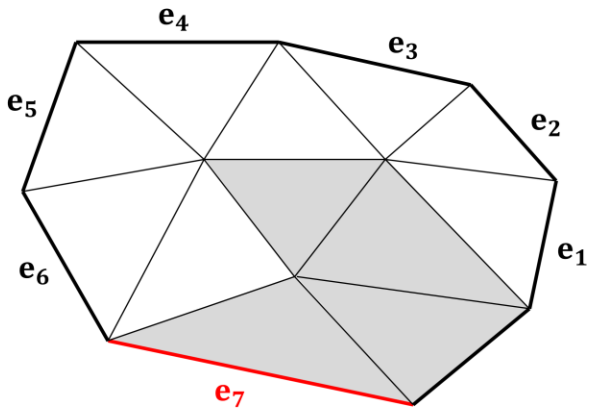
Quickhull 3D: Find Horizon



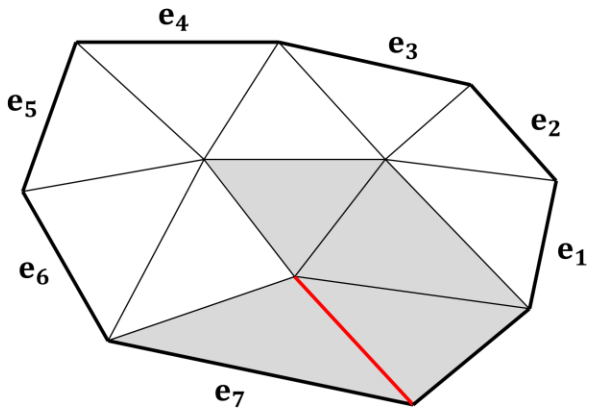
Quickhull 3D: Find Horizon



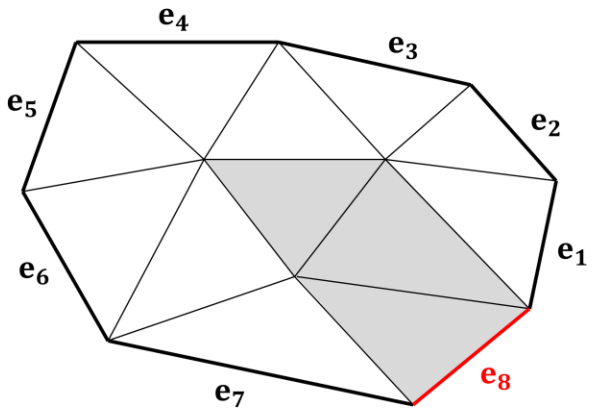
Quickhull 3D: Find Horizon



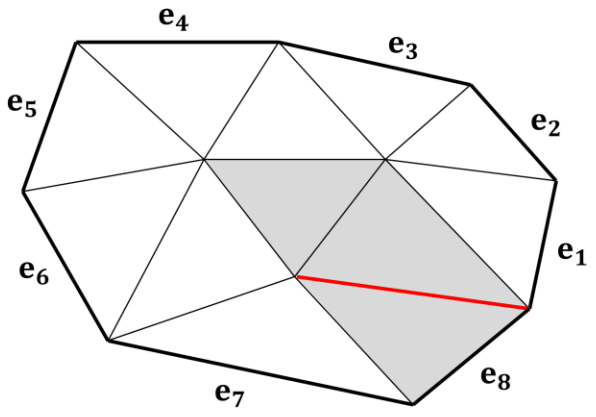
Quickhull 3D: Find Horizon



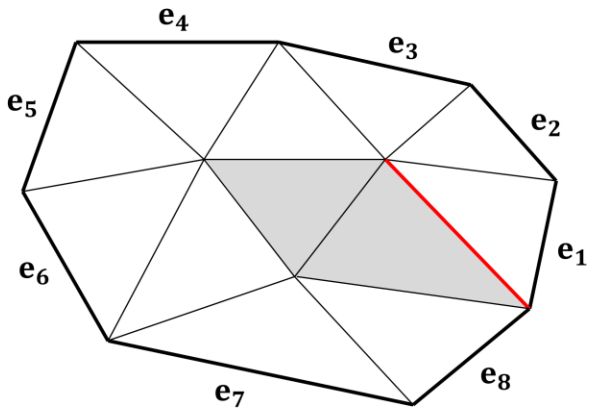
Quickhull 3D: Find Horizon



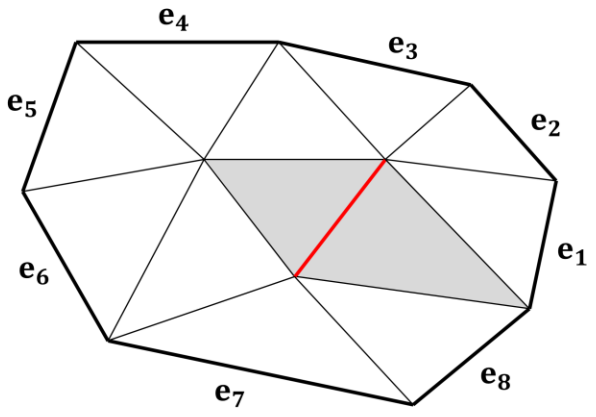
Quickhull 3D: Find Horizon



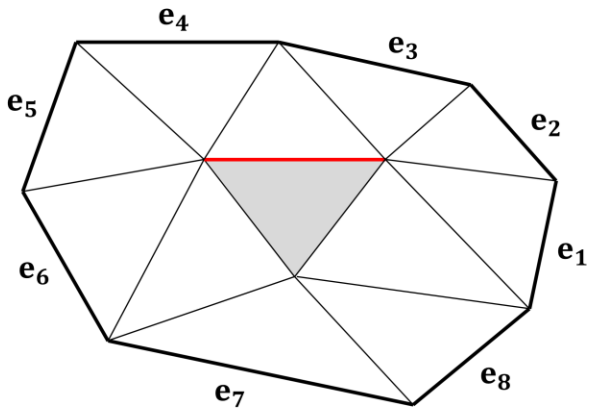
Quickhull 3D: Find Horizon



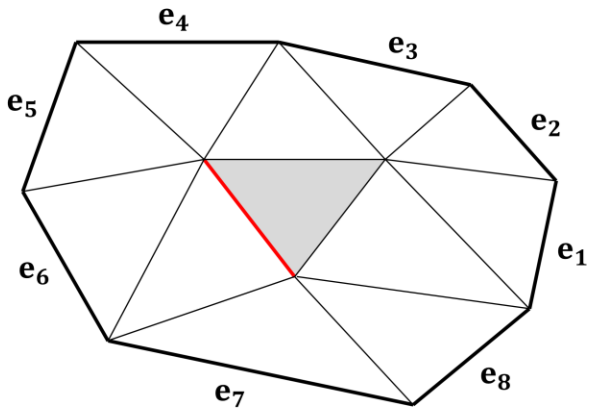
Quickhull 3D: Find Horizon



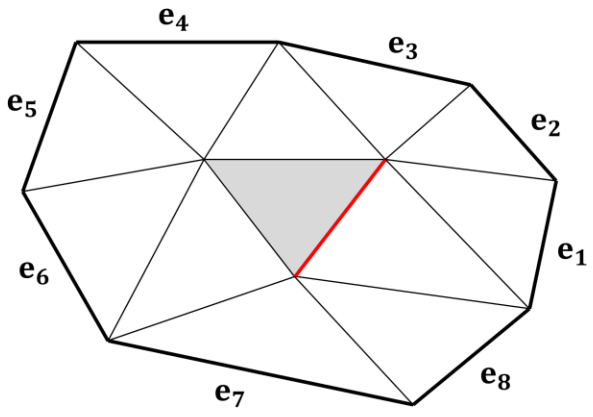
Quickhull 3D: Find Horizon



Quickhull 3D: Find Horizon

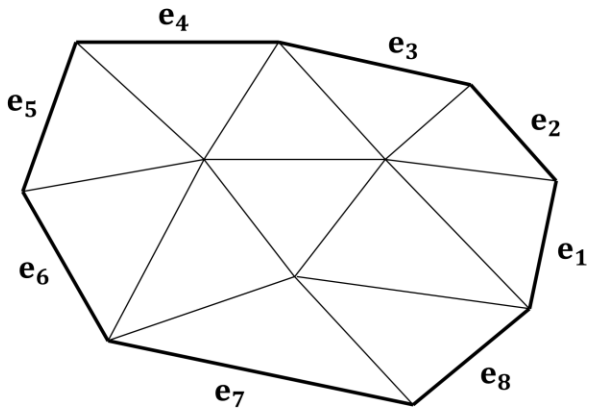


Quickhull 3D: Find Horizon



This is edge we started with and we are done!

Quickhull 3D: Find Horizon



On termination we have a list of all horizon edges in CCW order
- We now simply create a new triangle face for each horizon edge in our list
- This essentially connects the new vertex to the current hull

If you wonder why this works here is a quick answer:
- Every convex polyhedron can be represented by a planar graph and the horizon is essentially constructed performing a DFS on this graph.
- I don't want to go into graph theory here, but you might want to keep this in mind when you try to implement this function!

Let's now rewind and have see how this works again!

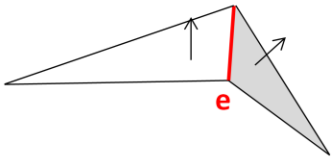
Outline

- Introduction
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- **Quickhull 3D Invariants**
- Implementation

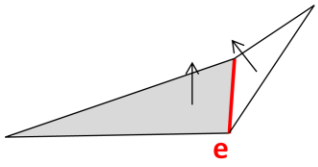
This closes the introduction of the 3D Quickhull algorithm
- As in 2D the next topic will be geometric and topological invariants we need to consider when constructing the hull

Quickhull 3D: Invariants

Edge **e** is convex



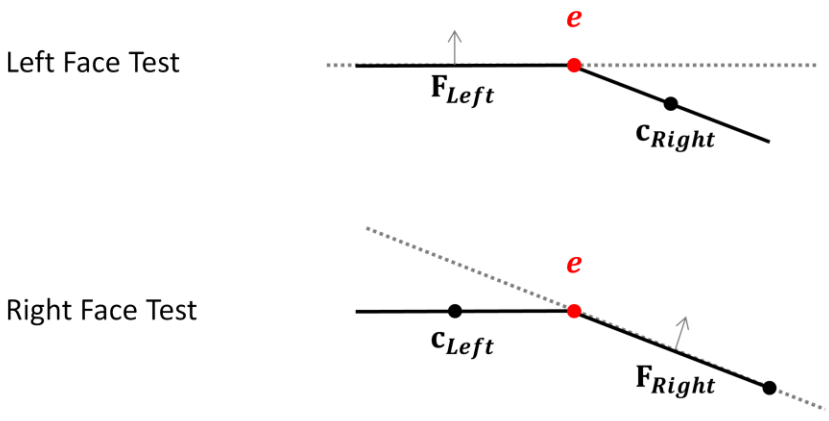
Edge **e** is concave



As in 2D we must maintain geometric invariants while building the hull:

- In 3D we must now guarantee that every **edge** is convex
- Again this should be obvious as well since otherwise it would be simple to find a line between two points inside the hull that would leave and enter as shown in the slide at the beginning of the talk

Quickhull 3D: Invariants



We need to define what a convex **edge** is and how we can test an **edge** for convexity:

For each edge:
- First we test if the center of the right face is below the left face plane
- The we test if the center of the left face is below the right face plane
- If both tests are true the edge is **convex**, otherwise it must be **concave** or **coplanar**
- This is very similar to the 2D test, but we now use the center point of the face
- The face center is simply the average of the face vertices

Quickhull 3D: Epsilon

$$\varepsilon = 3\left(\max_i\left((abs(x_i)\right) + \max_i\left((abs(y_i)\right) + \max_i\left((abs(z_i)\right)\right) \cdot FLT\_EPSILON$$

As in 2D we need an epsilon to define the flat planes and we just expand our formula to 3D!

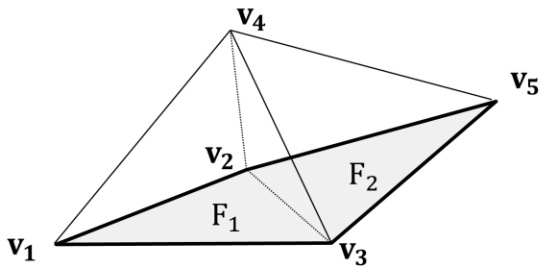
Quickhull 3D: Merging Coplanar Faces

Coplanar Faces:

$F_1 = \{\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}\}$

$F_2 = \{\mathbf{v_5}, \mathbf{v_3}, \mathbf{v_2}\}$

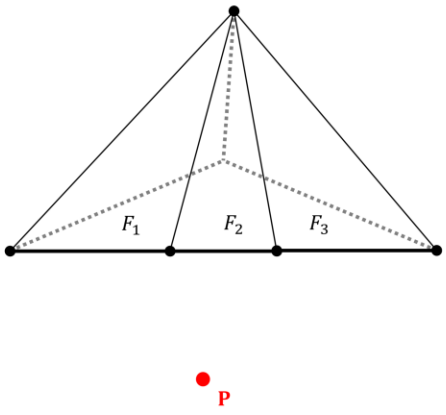
$F_{12} = \{\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_5}, \mathbf{v_3}\}$



So in 3D we now merge faces across non-convex edges. This add another step to our iterative loop:

- Let's assume face1 and face2 were coplanar in our previous example
- We would now merge face1 and face2 into a new **polygonal** face replacing the original faces F1 and F2
- As mentioned in the beginning we are not restricting ourselves to triangle faces

A Troublesome Hull



Situation: F_1 and F_3 are visible from new point $\mathbf{P}$ and F_2 is not!

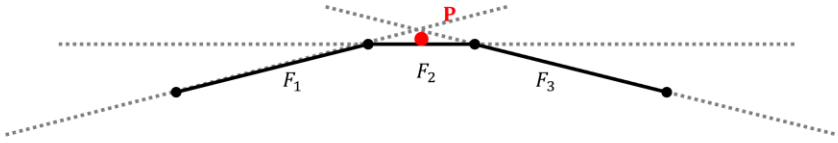
Let's now investigate the example from the original Quickhull paper:

- This is an example what happens if you DON'T merge faces and how you run into a bunch of geometrical and topological problems

Situation here: We have a new point $\mathbf{P}$. The faces F_1 and F_3 are visible from the new point $\mathbf{P}$ while face F_2 is not!

- Think of a tetrahedron and the front face is in the screen plane
- The front face was not merged into one big triangle face, but it is essentially a fan of three faces which are coplanar sharing non-convex edges
- Let's look at this example from the bottom and exaggerate the situation a bit

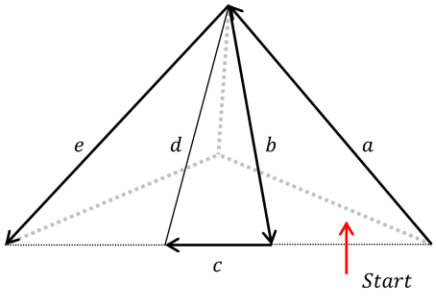
A Troublesome Hull (2)



Assume we are below point P and look UP:

- You can see the new point and the bottom edges of the three faces
- Since we didn't merge faces we might have potentially introduced non-convex edges
- You hopefully can see now how we could potentially end up in the situation described in the previous slide
- Of course the figure is highly exaggerated and doesn't need to be this extreme to go bad

A Pinched Horizon

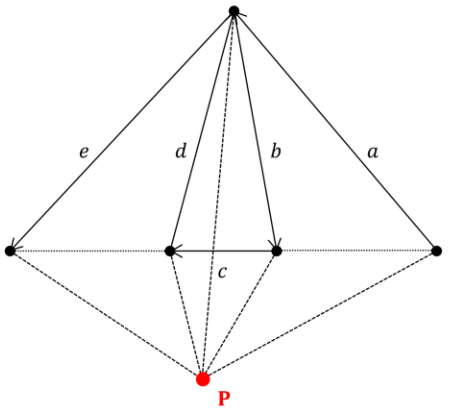


The depth – first traversal yields the horizon in CCW order

We now build the horizon as we learned earlier to find the horizon edges

- The results in five horizon edges a, b, c, d, and e
- Note how the horizon now pinches into the current hull at face1 and face3

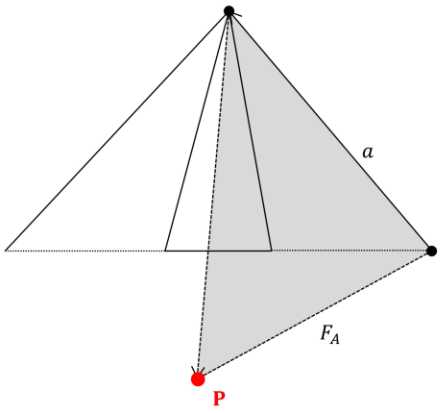
New Faces



We replace F_1 and F_3 with 5 new faces for each edge a−e

We replace F_1 and F_3 with 5 new faces for each edge a−e
- Let's look at each of the new faces individually and see what happens!

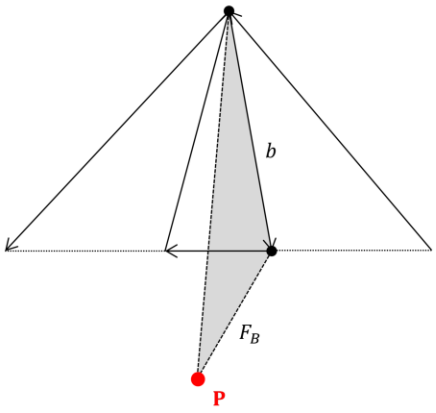
Face A



We build a face for edge a

- Note how this face partially overlaps face2 which is still on the hull

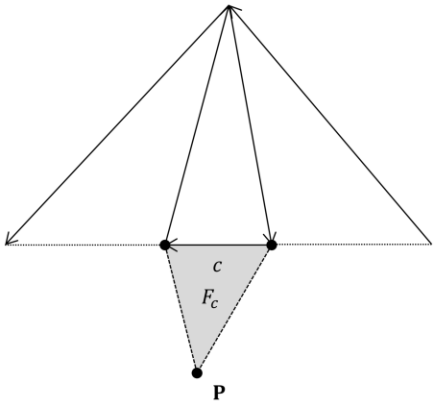
Face B



We build a face for edge b

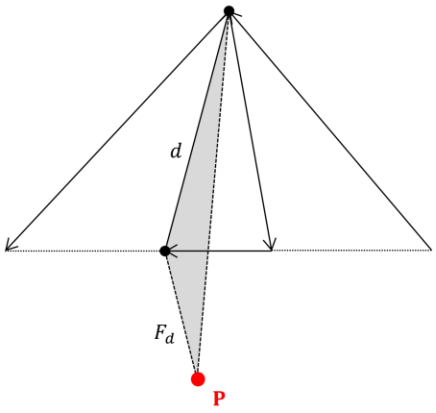
- Note that this face has flipped (CW) orientation and also shares and edge with FA

Face C



We build a face for edge c
- Nothing wrong here

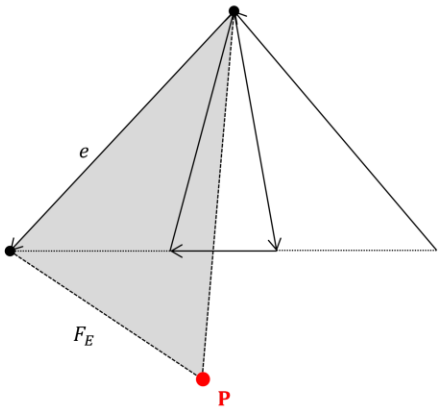
Face D



We build a face for edge d

- Note that this face has flipped (CW) orientation again and also shares an edge with FA and FB

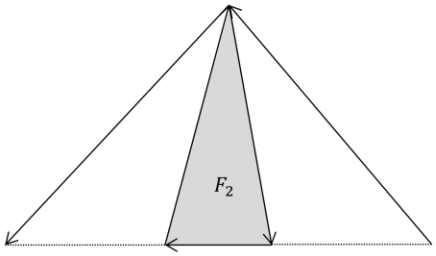
Face E



We build a face for edge e

- We now have four edges sharing the same edge and also partially overlap each other

Face 2



F_2 is still on the hull as well because it cannot be seen from the new point!

- Due to numerical imprecision F2 is still identified as visible from P as well

Errors

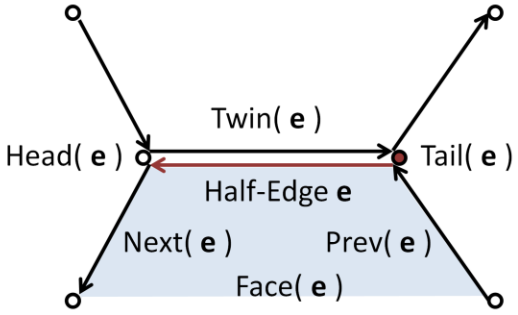
- Geometrical Errors
  - Two faces are flipped upside-down (F_B, F_D)
- Topological Errors
  - Four faces share the same edge (F_A, F_B, F_D, F_E)

As you probably noticed we introduced a couple of errors because our hull was not in an healthy state when we started adding the new point.

As a result the new faces are violating a bunch geometrical and topological invariants:
- Two faces are flipped upside down (that means the normals are pointing inside)
- Four faces share the same edge (which yields them partially and fully overlapping each other)

As you can imagine repairing those errors would become pretty involved.
The good news are that I haven't run into any of the described issues when properly merging faces during the hull construction.
So hopefully you can see that it essential to our implementation to maintain a healthy hull during construction.

Half-Edge Mesh



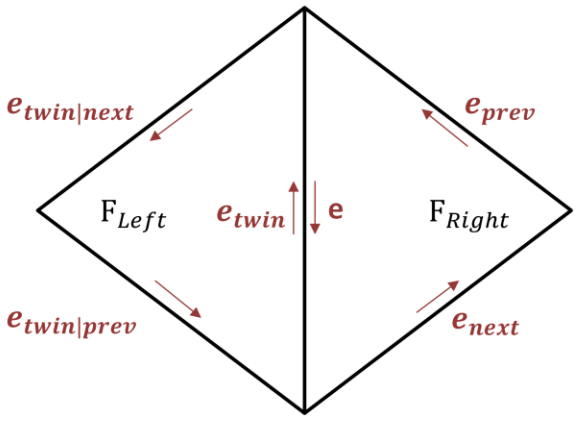
We haven't talked about a data structure for convex polyhedra yet. So before we start looking into face merging in more detail, let's talk about a possible data structure first:

- Obviously there are many ways to describe a convex polyhedron
- A common data structure is the so called Half-Edge data structure which is an edge centric mesh representation
- The half edge data structure makes it easy to iterate edges of a face and to access neighboring faces

For each face we store:
- A half-edge that defines the entry into a circular list around the face

For each edge we store:
- The previous and next edge that build the circular face list around the face
- A twin edge to cross over to adjacent faces
- The tail vertex of the edge
- Note that we don't need to store the head vertex since it is simple the tail vertex of the twin edge

Face Merging



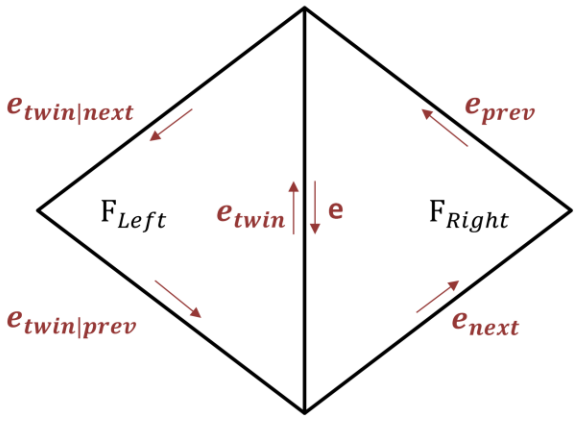
Set edge reference:
e->face->edge = e->prev

Absorb face:
e->twin->prev->face = e->face
e->twin->next->face = e->face

Link edges:
e->prev->next = e->twin->next
e->next->prev = e->twin->prev
e->twin->prev->next = e->next
e->twin->next->prev = e->prev

We learned that face merging is the essential operation to maintain a healthy hull.
- Here is an example how to merge two faces using the half-edge data structure.
- The situation is that we are about to merge the left into the right face:

Face Merging



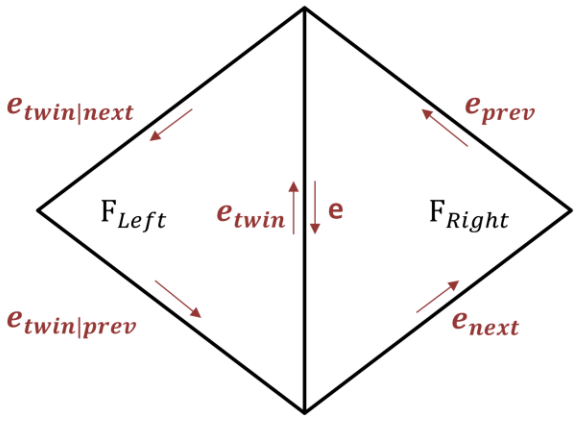
Set edge reference:
e->face->edge = e->prev

Absorb face:
e->twin->prev->face = e->face
e->twin->next->face = e->face

Link edges:
e->prev->next = e->twin->next
e->next->prev = e->twin->prev
e->twin->prev->next = e->next
e->twin->next->prev = e->prev

First we make sure that the absorbing right face does not reference the edge we are about to delete (e.g. we use edge->prev here)
- Avoiding those kinds of dangling pointers is actually what makes the implementation kind of interesting

Face Merging



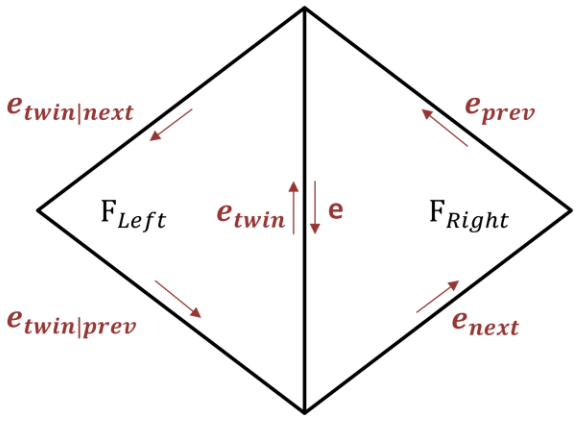
Set edge reference:
e->face->edge = e->prev

Absorb face:
e->twin->prev->face = e->face
e->twin->next->face = e->face

Link edges:
e->prev->next = e->twin->next
e->next->prev = e->twin->prev
e->twin->prev->next = e->next
e->twin->next->prev = e->prev

Next we must make sure that all edges of the absorbed left face new reference the absorbing right face

Face Merging



Set edge reference:
e->face->edge = e->prev

Absorb face:
e->twin->prev->face = e->face
e->twin->next->face = e->face

Link edges:
e->prev->next = e->twin->next
e->next->prev = e->twin->prev
e->twin->prev->next = e->next
e->twin->next->prev = e->prev

Finally we need to connect the incoming and outgoing edges at the deleted edge vertices

- I give you some time to have a closer look at this slide since it has a bunch of information

Face Merging (2)

- After a successful merge:
  - Remove merged face from face list
  - Delete merged edges
  - Delete merged face

After successfully merging two faces we can clean-up some things:

- We can remove the merged face from the global face list
- We then destroy merged edges
- Finally we can destroy the merged face

Face Merging: Newell Plane

- The Newell algorithm builds a best fit plane in the least square sense
- Sadly the computed normal might sometimes have only a few digits of precision
- We can improve precision by moving any vertex of the polygon into the origin

Face merging will create polygonal faces and we need to compute a face plane:
- The Newell method is common technique to build the plane for a coplanar polygon
- It builds a best fit plane in the least square sense
- Sadly the computed normal might sometimes have only a few digits of precision
- We can improve precision by moving any vertex of the polygon into the origin

Hint:
- See E. Catto's excellent blog post "A troublesome triangle" about this problem!

Face Merging: Newell Plane

$$n_x = \sum_{i=1}^{N} (y_i - y_{i+1}) \cdot (z_i + z_{i+1})$$

$$n_y = \sum_{i=1}^{N} (z_i - z_{i+1}) \cdot (x_i + x_{i+1})$$

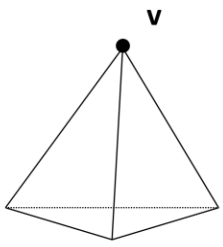
$$n_z = \sum_{i=1}^{N} (x_i - x_{i+1}) \cdot (y_i + y_{i+1})$$

$$\mathbf{p} = \frac{1}{N} \sum_{i}^{N} \mathbf{v_i}$$

I added some formulas for completeness and convenience, but going into detail here would get us off topic.

Gino's and Christer's book both cover Newell planes and you will also find good information using Google!

Topological Invariants

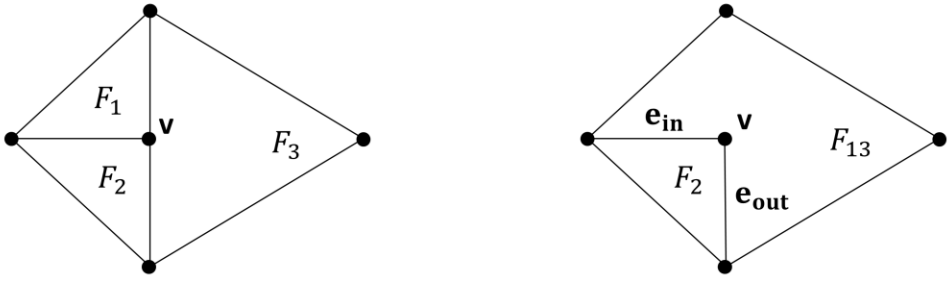


Each vertex must have at least three adjacent faces

Now let's look at one important topological invariant of a convex hull.

- Each vertex must have at least three adjacent faces
- Or three leaving edges if you prefer this view

Fixing Topological Errors (1)



When merging faces we might violate topological invariants and need to fix those again:

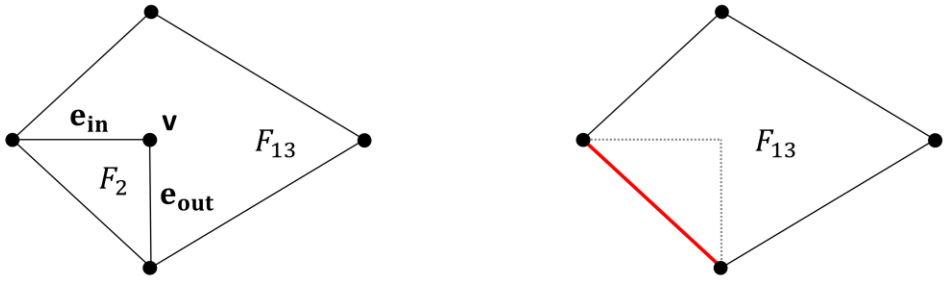
Consider the merge sequence in the above picture and how it can lead to topological errors:

- We merge face3 into face1

This creates a couple of problems:

- Vertex v has now only two adjacent faces
- The incoming and outgoing edges of v share the same face (F2) and are not distinct

Fixing Topological Errors (2)

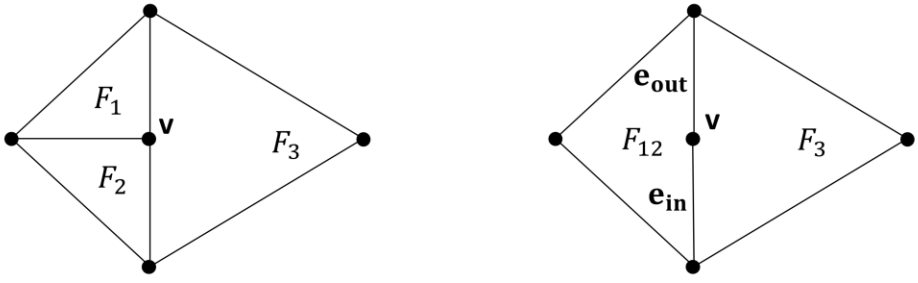


We detect this error by checking the adjacent faces of the in- and outgoing edges -> Both edges point to face2

- Since face2 is a **triangle** we don't need to connect the in- and outgoing edge, but use the non-shared edge instead
- Face2 is also redundant since all vertices are contained in face13 and can be deleted
- Vertex v has also become obsolete and should be deleted as well

When we merge faces we check for this error and fix it immediately!

Fixing Topological Errors (3)

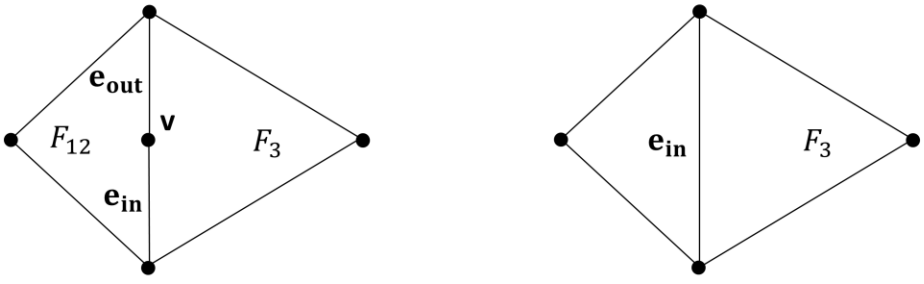


Let's look at the previous example again.

Consider now the slightly different merge sequence and how it leads to the same topological errors, but requires a slightly different fixing strategy:
- We now merge face2 into face1

Again:
- Vertex v has only two adjacent faces
- The incoming and outgoing edges of v share the same face (F3) and are not distinct

Fixing Topological Errors (4)

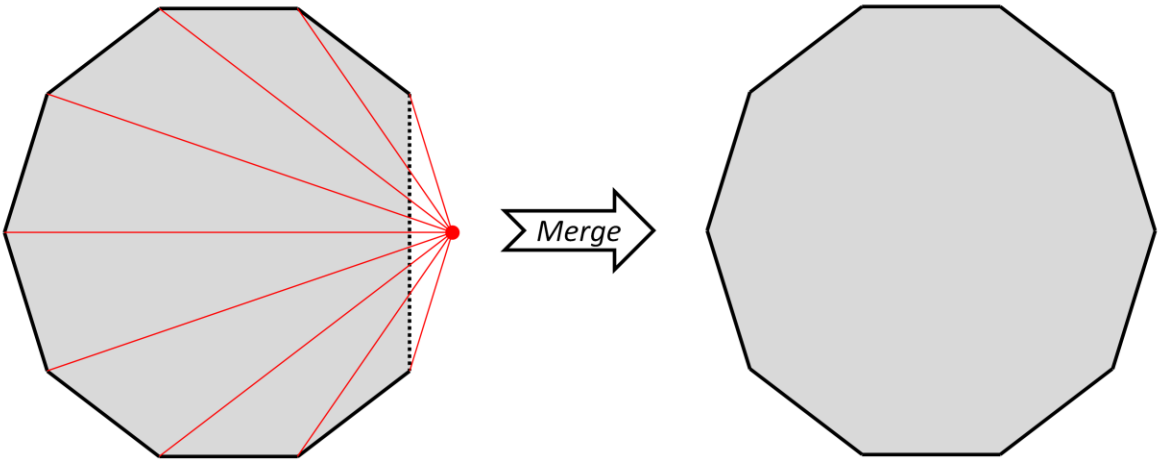


We detect this error again by checking the adjacent faces of the in- and outgoing edges -> Both edges point to face3 here

- Since face3 has more than three vertices we cannot apply the same fixing strategy as before
- Instead we extend the incoming edge to the next vertex and delete the outgoing edge
- Again vertex v has become obsolete and should be deleted as well

When we merge faces we check for this error and fix it immediately!

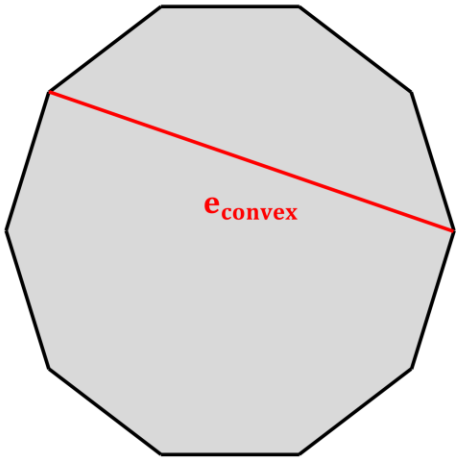
Large Merge Cycles



Imagine we are building the convex of hull of a cylinder and we are about to add the final vertex of the top face.

- This vertex is of course in the same plane as the other vertices.
- In this situation we create many new faces which are coplanar and need to be merged
- Ideally we would like to merge all new faces into one face as shown on the right hand side

Large Merge Cycles (2)



The problem is now that we merge one face after the other

- And whenever we merge two faces we rebuild the face plane.
- Rebuilding the face can jiggle the plane and an edge between two faces can become temporarily convex preventing us from merging the whole cycle
- It becomes e
- In the worst case this can introduce redundant or degenerate faces which are now **NOT** merged properly

Here are some ideas how to address this:

- The faces with the largest area should be the most stable w.r.t. the orientation of the face plane. So merging into the largest faces first reduces jiggle.
- You can also introduce an absolute tolerance to increase your merge radius and make your merge cycle be less sensitive for these situations.
- This is basically how I handle this problem at the moment since for physics we want as many large faces as possible for stability reasons and do not aim for the tightest hull.
- If you are working with collision margins it is probably a good idea to make the absolute tolerance a small percentage of that margin

Large Merge Cycles (2)

Another idea to deal with this problem is to **NOT** rebuild the face planes at all when merging to faces:

- For both faces you have your best fit plane and the vertices (which I tried to sketch on the slide)
- You can now compute the absolute distance of the right face's vertices to the left face's plane and vice versa
- Then simply keep the face plane that minimizes the distance instead of rebuilding it

Face Merging Strategies

- At each iteration we assume to start with a correct hull
- When adding the vertex we need to inspect all new faces for possible defects
- An easy strategy is to simply iterate all new faces and inspect their edges until we repaired all possible defects

Let's now talk how we could use face merging to deal with defect hulls:

- At each iteration we assume to start with a healthy hull
- When adding the vertex we need to inspect all new faces for possible defects at their edges between each other and at the horizon
- An easy strategy is to simply iterate all new faces and inspect their edges and repair defects one by one until we repaired all possible defects

Outline

- Introduction
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- **Implementation**

This closes the theory and in the remainder of this talk I like to share some tips about a possible implementation

Memory Bounds

- Worst case is that all input vertices are on the hull
- The number of vertices is then at most V = N
- The number of edges is then at most E = 3N – 6
- The number of faces is then at most F = 2N – 4
- Verify with Euler's formula: V – E + F = 2

  N – (3N – 6) + (2N – 4) = N - 3N + 6 + 2N – 4 = 2 (ok)

The major performance pitfall is bad memory management of the half-edge data structure:
- The convex hull for N vertices is bounded
- Worst case is that all input vertices are on the hull:
- The number of vertices is then at most V = N
- The number of edges is then at most E = 3N – 6
- The number of faces is then at most F = 2N – 4
- Test with Euler's formula: V – E + F = 2

We can pre-allocate one buffer for vertices, half-edges, and faces and manage this buffer in a free list
- Ideally we will just have one big allocation per hull construction!
- This becomes especially important if you plan to compute convex hulls at runtime (e.g. for destruction)

Some practical details:
- Don't forget that you need to allocate half-edges (twice the number of edges)
- Also account for temporary allocations (e.g. horizon faces)
- In practice just double the buffer size

Implementation : Half-Edge Mesh (1)

```
struct qhVertex
{
  qhVertex * Prev;
  qhVertex * Next;

  // Optional
  qhHalfEdge* Edge;

  qhVector3 Position;
};
```

Let's start with the vertex structure.

- I am using an intrusive list to store the vertices in the global vertex list of the hull or in a conflict list
- So obviously we have to include the list pointers here.
- You can optionally also store an edge leaving the vertex.
- This is not needed for constructing the hull, but it can be useful for post-processing if you like e.g. to iterate all adjacent faces of the vertex
- Of course we also to need store the position of a vertex

Implementation: Half-Edge Mesh (2)

```
struct qhHalfEdge
{
   qhVertex* Tail;

   qhHalfEdge* Prev;
   qhHalfEdge* Next;
   qhHalfEdge* Twin;

   qhFace* Face;
};
```

Now let's have a quick look how we can potentially implement a half-edge.

- As one would expect this definition maps pretty directly to a possible data structure
- We store a pointer to the tail vertex of the edge
- The edges build a circular list around the face so we also need to store the list pointers
- We also store the twin edge to cross over to adjacent faces
- Finally we also keep a reference to the parent faces of the edge

Implementation : Half-Edge Mesh (3)

```
struct qhFace
{
   qhFace* Prev;
   qhFace* Next;

   qhHalfEdge* Edge;
   qhList< qhVertex > ConflictList;
};
```

Finally the face structure.

- I am also using an intrusive list here, so we have to include the list pointers again
- And of course there is a pointer to the first edge starting the circular list around the face
- And of course we can store our conflict list here as well

Implementation: Construction

```
void qhConvex::Construct( const qhArray< qhVector3 >& Vertices )
{
    if ( !BuildInitialHull( Vertices) )
        return;

    qhVertex* Vertex = NextConflictVertex();
    while ( Vertex != NULL )
    {
        AddVertexToHull( Vertex );
        Vertex = NextConflictVertex();
    }
};
```

Finally some high-level code examples to give you an idea of a possible implementation:

- Assume we have some qhConvex class to store the hull after construction
- This snippet shows the top level construction function

Explain a bit…

Implementation: Iteration

```
void qhConvex::AddVertex( const qhVertex* Vertex )
{
    qhArray< qhHalfEdge* > Horizon;
    BuildHorizion( Horizon );

    qhArray< qhFace* > NewFaces;
    BuildNewFaces( NewFaces, Horizon );
    MergeFaces( NewFaces );

    ResolveOrphans( NewFaces );
};
```

The code snippets shows the iterative AddVertex() function

- We add new points until our conflict lists are empty

Explain a bit…

Dude, where is the code?

- QHull (C/C++)
- www.qhull.org

- Quickhull3D (Java)
- http://www.cs.ubc.ca/~lloyd/java/quickhull3d.html

So you want to implement Qhull yourself and I talked now for nearly an hour and there is no code!

- Luckily there is a beautiful open-source implementation in JAVA which you can use to start
- I also recommend looking at the original Qhull implementation which is also a great implementation and full of gems of computational geometry!!!

Hopefully my presentation will help you to understand and implement a robust convex hull builder!

Thanks!

- Thanks to Valve
- Thanks to Paul, Steve, Jeff, Gurjeet, Paul, Bruce, Shannon and Anoush for rehearsing
- Thanks to Randy for reading the presentation at an early stage and providing valuable feedback
- Special thanks to Erin for endless discussions about Quickhull
- Special thanks to John Lloyd for sharing his beautiful JAVA Quickhull implementation and making it open source

Before I close I like to thank a bunch of people!

- Thanks to Valve for giving me permission to present to you today
- Thanks to Paul, Steve, Jeff and Anoush for spending time and rehearse this presentation with me
- Thanks to Randy for reading the presentation several times at an early stage and providing valuable feedback
- Special thanks to Erin for endless discussions about Quickhull
- Special thanks to J. Lloyd for sharing his beautiful JAVA Quickhull implementation

Questions

I prepared a little demo which I like to show first and then I will answer questions!
Thank you!

References

- G. v. d. Bergen: “Collision Detection in Interactive 3D Environments”
- C. Ericson: “Real-time Collision Detection”
- G. Rhodes: “Computational Geometry” (Half-Edge Mesh)
- M. McGuire: “The Half-Edge Data Structure” (FlipCode)
- E. Catto: www.box2.org (Newell Plane)