Final exam

Duration : 3 hours.

Foreword

Every answer deserves a justification, never simply answer yes or no unless instructed otherwise. Algorithms should be provided as pseudo-code in no specific language. This exam is made of several independent parts and questions, take your time going over it before you dive into a particular section, to choose depending on what seems easier for you. Avoid getting stuck and do no hesitate to leave blank answers.

1 On meshes

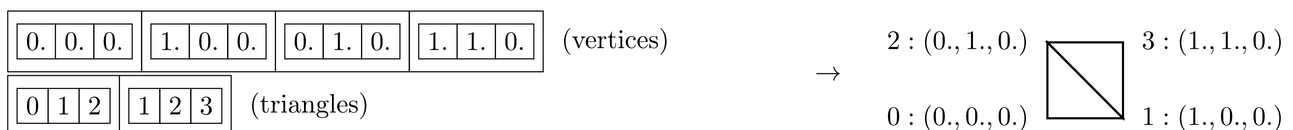
1.1 Orientation of indexed array meshes

This part deals with mesh data structures to represent 3D objects. In all the following we only deal with *triangular* meshes, whose faces are all triangles.

An indexed array mesh describes a surface using two arrays storing vertices and triangles :

- the vertex coordinates as triplets of floating point numbers ;
- the triangles as triplets of integer indices, referencing the vertices stored in the vertex array.

You can find below an example of an indexed array mesh with two triangles and four vertices.



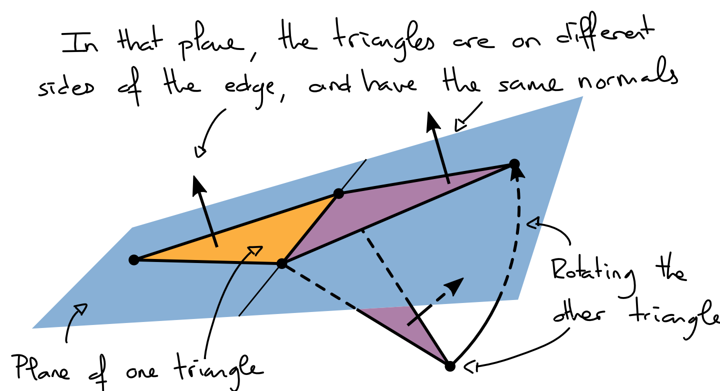
Orienting a triangle consists in defining an *out* side and an *in* side for the triangle. Given a triangle with vertices $\mathbf{v}_0$, $\mathbf{v}_1$ and $\mathbf{v}_2$ (in that order in the array), let $\mathbf{n}$ be the normal of the triangle computed as

$$\frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{\|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\|}.$$

By convention the *out* side of the triangle is the side where $\mathbf{n}$ lies.

Question 1 – Compute the normals of the two triangles in the above example.

Two neighboring triangles are said to be *consistently oriented* if they agree on their outside. Roughly, if you were to paint the outside of one of the triangles, and spilled paint over their common edge, you would end on the outside of the other triangle. More formally, the two triangles are consistently oriented if one of the triangles can be rotated with its normal around their common edge to the plane of the other triangle, such that the triangles are on different sides of the edge, and their normals are the same.

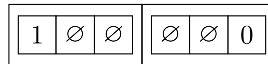


Question 2 – In the above example, are the two provided triangles consistently oriented ?

Question 3 – Write an algorithm to rearrange an indexed array mesh in order to flip the orientation of one (and only one) of its triangles, swapping its outside and inside. You can refer to it as flip(t) from now on.

Question 4 – Does the consistent orientation of two neighboring triangles actually depend on the vertex positions ? Write the algorithm for a quick test to check that consistency. You can refer to it as consistent(t_1,t_2) from now on.

A mesh is consistently oriented if any two neighboring triangles in the mesh are consistently oriented. The following questions aim at obtaining such a mesh. To be able to navigate through the mesh, we augment the indexed array mesh structure with another array of neighbors. For each triangle, its neighboring triangles are listed as a triplet of triangle indices. If the triangle has vertex indices v_0, v_1 and v_2 in that order, and neighbor indices t_0, t_1 and t_2 in that order, then t_0 is the index of the neighboring triangle opposite to v_0, sharing vertices v_1 and v_2, t_1 is opposite to v_1 and t_2 is opposite to v_2. When no neighboring triangle exists along an edge, a special index ∅ is used. In the previous example, the neighbor array would be :



1	∅	∅	∅	∅	0

since triangles 0 and 1 are neighbors, sharing the vertices 1 and 2. We also say that a mesh is connected if any triangle can be reached from any other triangle using neighborhood relationships.

Question 5 – Given a connected indexed array mesh with neighbors, and using classical data structures (lists, arrays, stacks, queues, dictionaries), write an algorithm which consistently reorients a given indexed array mesh. You can refer to it as reorient(M) from now on.

Question 6 – (Bonus) Is it always possible, given an input mesh, to reorient it consistently ? If not, propose a counter example, and describe a test to check that in your reorient algorithm.

When a mesh encodes the boundary of an object, there are no boundary edges in the mesh, and it is classical to ensure that the outside of the triangles actually face the outside of the object.

Question 7 – Given an indexed array mesh corresponding to the surface of an object, provide a method to get a point in space that is outside the object.

Question 8 – Suppose you have a function raytrace(M, $\mathbf{p}$, $\mathbf{d}$) providing the index of the first triangle in the mesh M hit by a ray emitted from a point $\mathbf{p}$ in direction $\mathbf{d}$. Provide a test to determine whether the outsides of the triangles in a coherently oriented mesh match the actual outside of the corresponding object.

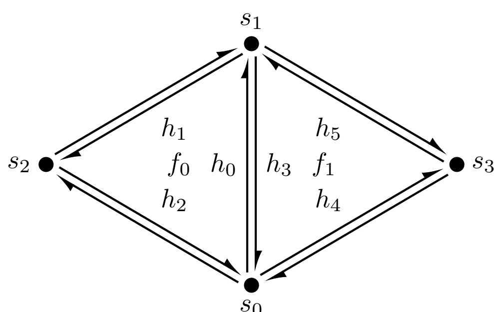
1.2 Halfedge meshes

Another classical data structure to store meshes is the halfedge data structure. We are only considering triangle meshes here. In such a structure, three types of elements are stored :

Vertex :
- point : an array of three floats (coordinates) ;
- halfedge : a halfedge sarting at the vertex.

Face :
- halfedge : a halfedge bording the face.

Halfedge :
- vertex : the vertex at the tip of the halfedge ;
- face : the face borded by this halfedge ;
- opposite : the halfedge bording the neighboring face along the same edge ;
- next : the halfedge starting at the tip of this halfedge on the same face.



- s_0.halfedge $= h_0$ or h_4 or any other outgoing ;
- s_1.halfedge $= h_1$ or h_3 or any other outgoing ;
- f_0.halfedge $= h_0$ or h_1 or h_2 ;
- f_1.halfedge $= h_3$ or h_4 or h_5 ;
- h_0.vertex $= s_1$;
- h_0.face $= f_0$;
- h_0.next $= h_1$;
- h_0.opposite $= h_3$;

The tip of a halfedge is always the source of its next halfedge, and the source of its opposite halfedge. Some examples of algorithms :

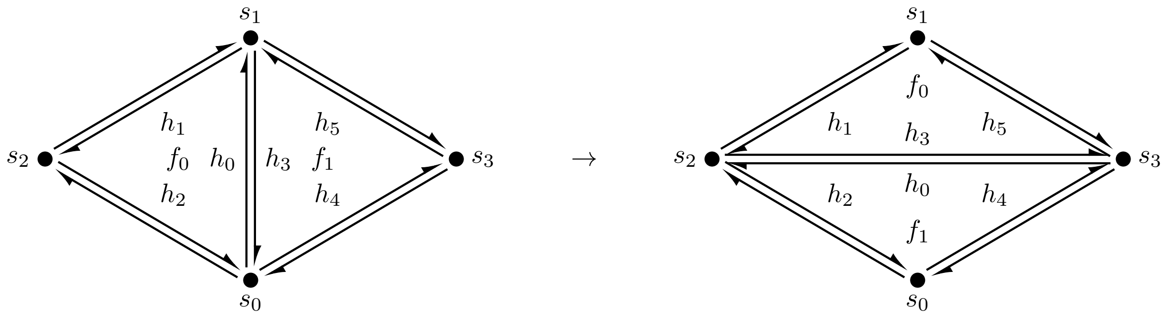
```
Function edgeVertices(h) → pair of vertices
    // returns both vertices of a halfedge
    return (h.vertex, h.opposite.vertex)
```

```
Function faceVertices(f) → set of vertices
    // returns all the vertices of a face
    S ← ∅, begin ← f.halfedge, cursor ← begin
    repeat
        insert cursor.vertex into S
        cursor ← cursor.next
    until cursor == begin
    return S
```

Question 9 – Write an algorithm returning the set of faces neighboring a given face.

Question 10 – Write an algorithm returning the set of vertices neighboring a given vertex.

A classical operation to improve the quality of a triangle mesh is an *edge flip*. The flip transforms two neighboring triangles along a given edge as follows :



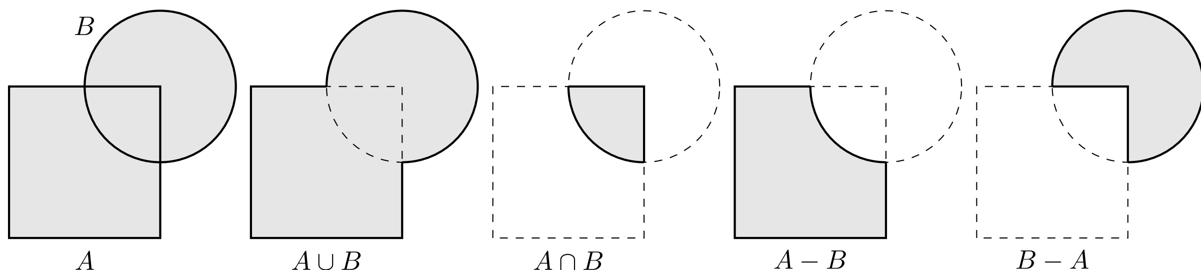
Question 11 – Write an algorithm performing an edge flip given one of the halfedges along the flipped edge between two neighboring triangles.

Question 12 – Write an algorithm to convert an input indexed array mesh with face neighbors into a halfedge mesh. You can use a `create` function to create vertices, faces and halfedges, returning a reference to the created element.

2 Ray Tracing Constructive Solid Geometry

Constructive Solid Geometry is a representation for 3D objects that is very common in computer assisted design, where the goal is to describe mechanical pieces and objects that are going to be fabricated for real. This representation combines simple primitive shapes (boxes, spheres, cylinders, ...) into more complex ones using three operations. Given two shapes A and B, a point belongs to

- *the union* $A \cup B$ if it belongs to A or to B (or to both) ;
- *the intersection* $A \cap B$ if it belongs to both A and B ;
- *the difference* $A - B$ if it belongs to A but not to B ;



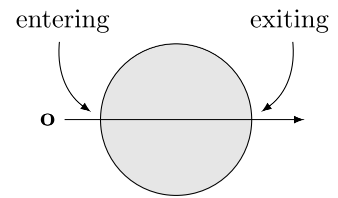
This section aims at ray tracing objects described by such operations.

2.1 Tagging intersections on a single object

In the following, we consider a ray described by an origin $\mathbf{o}$ and a normalized direction $\mathbf{d}$. Any point $\mathbf{p}$ on the ray can be described as $\mathbf{p} = \mathbf{o} + a\mathbf{d}$ with $a \in \mathbb{R}^+$. Given an object A, we define the sequence of its intersections with the ray as $I_A = [a_1, a_2, \ldots, a_n]$ with $a_1 < a_2 < \ldots < a_n$. Note that we only consider here objects for which this sequence is *strictly* increasing. For simplicity reasons, we also consider that ray / object intersections are always *generic* : this means that we ignore special cases like when a ray intersects an object tangentially or any

other such highly unlikely event. Such a hypothesis can often be simulated using symbolic perturbation. An intersection can therefore always be tagged as *entering* or *exiting* an object.

An intersection $\mathbf{p} = \mathbf{o} + a\mathbf{d}$ between the ray and the object is *entering* if there exists some $\varepsilon > 0$ such that $\mathbf{p} + (a - \varepsilon)\mathbf{d}$ is out of the object and $\mathbf{p} + (a + \varepsilon)\mathbf{d}$ is in the object.



An intersection $\mathbf{p} = \mathbf{o} + a\mathbf{d}$ between the ray and the object is *exiting* if there exists some $\varepsilon > 0$ such that $\mathbf{p} + (a - \varepsilon)\mathbf{d}$ is in the object and $\mathbf{p} + (a + \varepsilon)\mathbf{d}$ is out of the object.

Question 13 – What is the tag of the *last* intersection between the ray and a bounded object ?

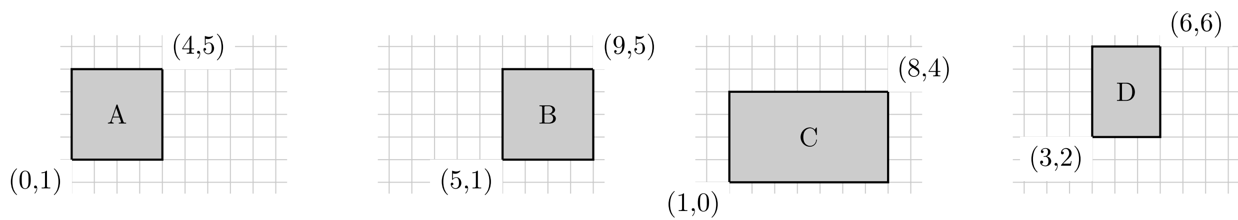
Question 14 – Write an algorithm to tag every intersection between the ray and a bounded object.

Question 15 – Describe a test to check whether $\mathbf{p}$ is inside or outside the object.

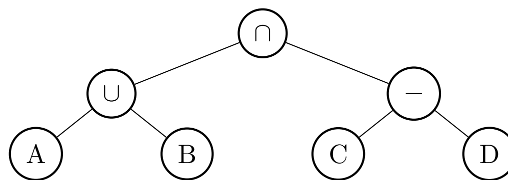
2.2 CSG trees

CSG objects are described by trees. The leaves are the primitive objects, and each node is an operation involving the children. Note that in the case of the difference, the order of the children matter, and the right child is subtracted from the left one.

Question 16 – Given the four following rectangles as primive objects :



draw the object resulting from the following CSG tree :



Question 17 – Draw the intersections between the resulting object and the ray originating at $(0, 3)$ with direction $(1, 0)$.

2.3 Induction

The goal in this section is to compute the intersection between a ray and an object defined by a CSG tree by induction, combining the intersections of the children to obtain those of the parent. We now consider two objects A and B with respective intersection sequences $I_A = [a_1, \dots, a_n]$ and $I_B = [b_1, \dots, b_m]$. We suppose that $a_i \neq b_j$ for any two intersections $a_i \in I_A$ and $b_j \in I_B$.

Question 18 – Let $a_i \in I_A$ be an intersection with A tagged as entering. Write an algorithm examining I_B to determine whether a_i is an intersection with $A \cup B$. What is the complexity of your algorithm as a function of m the number of intersections with B ?

Question 19 – Write an algorithm to compute $I_{A \cup B}$ given I_A and I_B. What is the complexity of your algorithm as a function of n and m the numbers of intersections in respectively A and B ?

Question 20 – Detail how you would modify the previous algorithm to obtain $I_{A \cap B}$ or $I_{A - B}$