Sarah Naja
Felix Becker
Philipp Hertkorn
Yunus Tan

Eberhard Karls
Universität
Tübingen

Programming in C++

Sheet 6

Submission date: 09.10.2024 12:00

Constructive Solid Geometry

In this exercise, you will implement Constructive Solid Geometry (CSG). CSG refers to the assembly of complex shapes from simple geometric primitives and is widely used e.g. in CAD applications.

The base code already includes a basic `Point3D` class, a class for Axis-Aligned Bounding Boxes `AABB`, and a `Shape` super-class which describes the "interface" of an abstract `Shape`. It is defined in `shapes.h`. Each shape needs to provide information about its bounding box and it needs to be able to tell whether any given point is contained in the shape.

To retrieve the bounding box of a shape, one can call:

```
AABB getBounds() const;
```

To check whether a point is included in a shape, one can call:

```
bool isInside(const Point3D& p) const;
```

The `Shape` class is simultaneously the base of all `Shape` implementations and also a wrapper around pointers to arbitrary shapes. This allows us to create `Shape` instances with arbitrary behavior, rather than constantly having to deal with pointers ourselves. These `Shape` instances defer all functionality to their instance pointer:

```
std::shared_ptr<const Shape> instance {nullptr};
```

Functionality of individual shapes needs to be implemented in the `virtual` implementation functions:

```
virtual AABB getBounds_impl() const;
virtual bool isInside_impl(const Point3D& p) const;
```

Never call these functions directly, they should only be called through the `Shape`'s public interface.

For the bounding box, the default implementation returns `AABB{-1.0f, 1.0f}`, which expands to `AABB{Point3D{-1.0f}, Point3D{1.0f}}` with `Point3D(float x) : Point3D{x, x, x} {}`. I.e, the default bounding box is constructed from the corners $(-1,-1,-1)^t$ and $(1,1,1)^t$. It coveres everything within the cuboid spanned by these two points. Derived shapes with different bounds need to provide their own `override`.

When describing which points belong to a shape, there is no sensible default. The implementation of the base `Shape` will simply throw an exception. Each derived `Shape` implementation *must* provide an `override` to the `isInside_impl` function. Rather than representing the shape explicitly (as a mesh would with points and faces), this function represents the shape implicitly. As two simple examples, the `Empty` shape returns `false` for all points and the `Cube` shape only checks whether the point is contained in the bounding box.

For the final visualization, an explicit representation is required. To do so, we use a simple voxel grid which consists of regular (almost) cubical cells which are either occupied or empty. To get any output, you will have to finish the implementation of the provided `VoxelGrid` class.

6.1 Shapes (40 Points)

C++

With the shapes given so far, we can only create cubes, which severely limits the types of composite shapes we can express.

a) In `shapes.h` and `shapes.cpp`, implement the shape classes `Sphere`, `Cylinder`, and `Octahedron` by deriving from `Shape`, similar to `Cube`. The sets of points belonging to these shapes are easily described as follows:

Sphere A sphere with radius 1, centered at the origin $(0, 0, 0)^t$:

$$\forall\, p \in \texttt{Sphere} \;:\; \|p\|_2 \leq 1 \;\Leftrightarrow\; \sqrt{p_x^2 + p_y^2 + p_z^2} \leq 1.$$

Cylinder A cylinder with radius 1 in xy and height 2 in z, centered at the origin $(0, 0, 0)^t$:

$$\forall\, p \in \texttt{Cylinder} \;:\; \sqrt{p_x^2 + p_y^2} \leq 1 \wedge -1 \leq p_z \leq 1.$$

Octahedron An octahedron with radius 1, centered at the origin $(0, 0, 0)^t$:

$$\forall\, p \in \texttt{Octahedron} \;:\; \|p\|_1 \leq 1 \;\Leftrightarrow\; |p_x| + |p_y| + |p_z| \leq 1.$$

In addition to the function for querying whether points are inside the shape

```
bool isInside_impl(const Point3D& p) const override;
```

you will also have to implement the clone function

```
Shape clone_impl() const override;
```

The purpose of this function is to create a deep copy of the derived shape type. Use `std::make_shared` to create a shared pointer to a copy of the derived shape type and pass it to the move constructor `Shape(std::shared_ptr<Shape>&& shape) noexcept`, like already implemented for the `Cube` class.

6.2 Shape Operations (40 Points)

C++

An operation between shapes is a function of one or two shapes that outputs another shape. The output shape is described by a boolean expression between the input shapes. A helpful super-class `Operation` is already provided. You need to implement the following operation sub-classes in `operations.h` and `operations.cpp`: `And`, `Or`, `Xor`, and `Not`.

And – the space where both shapes overlap.

Or – the space occupied by one or both shapes.

Xor – the space covered by exactly one of the two shapes.

Not – the space not covered by a shape. **Beware:** this operation only takes one shape!

a) Implement a constructor that forwards the input shape(s) to

```
Operation(const Shape& shape_a, const Shape& shape_b);
```

and the clone function

```
Shape clone_impl() const override;
```

b) Implement

```
bool isInside_impl(const Point3D& p) const override;
```

to return the result of the boolean expression applied to the input shape(s).

c) Implement the `public` boolean operators `Shape operator&(const Shape& other) const`, `Shape operator|(const Shape& other) const`, `Shape operator^(const Shape& other) const`, and `Shape operator!() const` in the `Shape` class. Return a `clone()` of the new `Operation` instance to always return a base `Shape`.

This will allow you to create new shapes by simply writing expressions involving existing shapes, e.g. `Shape difference = Square{}-Sphere{};`.

d) Implement `Shape operator+(const Shape& other) const` such that `a+b` returns $A \cup B$, and `Shape operator-(const Shape& other) const` such that `a-b` returns $A \setminus B$ (the space covered by A which is not covered by B) in the `Shape` class. You may use the other operators you just implemented to simplify this task.

6.3 Shape Transformations (60 Points)

C++

In order to manipulate the various 3D shapes, we need to define some 3D transformations. A helpful super-class `Transformation` is already provided. Here, we will tackle the following tasks:

Scaling Scale the shape at the origin in x, y, and z by the factors provided as a `Point3D` s:

$$\text{scale}(p, s) = \begin{pmatrix} p_x \cdot s_x \\ p_y \cdot s_y \\ p_z \cdot s_z \end{pmatrix}$$

Translation Move the shape in x, y, and z by the offset provided as a `Point3D` t:

$$\text{translate}(p, t) = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \end{pmatrix}$$

Rotation Rotate the shape at the origin around one of the primary axes. More precisely, given axis x, rotate around the yz-plane, given y, rotate around the zx-plane, and, given z, rotate around the xy-plane. Points on the 2D plane can be rotated using the rotation matrix:

$$\text{rotate}(p, \alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \end{pmatrix}.$$

The corresponding function `rotate2D` is already provided in `transformations.cpp`.

Think about how these operations transform the shape's bounding box and how you need to transform a query point provided to the `isInside` function, such that you can pass on the query to the nested shape. For the rotated bounding box, you may assume that the nested shape completely fills its bounding box. I.e., you can simply rotate all corners of the existing bounding box and compute a new bounding box from the rotated points.

a) In `transformations.h` and `transformations.cpp`, implement the classes `Scaled`, `Translated`, and `Rotated`. They shall all derive from `Transformation`, which in turn derives from `Shape`. Therefore, all transformations actually act as shapes and allow for further transformations (or operations) to be applied to them.

For each transformation class, implement the following functions:

```
Shape clone_impl() const override;

AABB getBounds_impl() const override;
bool isInside_impl(const Point3D& p) const override;
```

Also, add the necessary class members and initialize them during construction.

For the `clone_impl` function, note that you can pass constructor arguments to `std::make_shared`.

b) For convenient use of the transformations, add the following functions to the `Shape` super-class:

```
Shape scaled(Point3D factor) const;
Shape translated(Point3D offset) const;
Shape rotated(Axis axis, float angle) const;
```

Implement them to return a clone of the transformed shape.

6.4 Voxel Grid (60 Points (15+10+5+10+5+15+0))

C++

The `VoxelGrid` is used to produce the final output for visualization. Its primary task is the voxelization of arbitrary input shapes. The voxel grid adjusts to the space covered by its input shape and stores them in the member variable `bounds`. For each unit of space, a set amount of voxels needs to be allocated in each dimension. This is controlled by the compile-time constant `level_of_detail`:

```
const static uint32_t level_of_detail = 8;
```

The effective resolution is stored in `res_x`, `res_y`, and `res_z`. The individual voxels are compactly stored in the flat vector `std::vector<bool> voxels` which uses a single bit per entry. You have to determine how to store and access the voxels in the `std::vector<bool> voxels`.

a) Implement the helper function

```
Point3D voxelCenter(uint32_t x, uint32_t y, uint32_t z) const;
```

Given a voxel index in x, y, and z, return the 3D point of the voxel's center. Account for the position and size of the voxel grid's axis aligned bounding box (AABB). Voxels are not necessarily perfect cubes but rather evenly divide the voxel grid's bounding box into cuboids. The number of subdivisions in each dimension is given by `res_x`, `res_y`, and `res_z`.

First, compute the stepsize per subdivision in each dimension (essentially the size of one voxel cuboid) from the bounding box's extents. Then, multiply the stepsize by $(x + 0.5, y + 0.5, z + 0.5)$ to get the center location. Finally, add the bounding box's minimum location.

b) Implement the implicit conversion from `Shape` to `VoxelGrid`:

```
VoxelGrid(const Shape& shape);
```

Copy over the shape's bounding box using member initialization. Determine the resolution of the voxel grid by multiplying the shape's bounding box's extents with the `level_of_detail` constant. Round down by converting the result to an unsigned integer and ensure that the resolution in each dimension is at least one. (Make sure not to cast $\pm\infty$ to `int`, since that is undefined behavior.) Then, allocate space for the voxels by resizing `std::vector<bool> voxels`. Finally, loop over all voxels and set them if their center point (use the `voxelCenter` helper function) is contained in the input shape.

c) Implement the helper function

```
bool isSet(uint32_t x, uint32_t y, uint32_t z) const;
```

Its task is to simply check in the `voxels` vector, whether the corresponding bit is set.

d) Implement the slicing function

```
VoxelSlice extractSlice(Axis axis, uint32_t slice) const;
```

Its task is to construct a `VoxelSlice(uint32_t res_x, uint32_t res_y)` which can afterwards be used to print its contents to the console. A `VoxelSlice` is constructed from its x and y dimensions and consists of a nested vector

```
std::vector<std::vector<bool>> data;
```

The outer vector will have the size of the y dimension and the inner vectors will have the size of the x dimension. You can use `data[y][x]` to access bits in the nested vectors.

Of course, this only applies to x and y if the `axis` is `Axis::Z` and you are creating a slice containing the xy-plane. When slicing given the x coordinate, you need to create a slice in the yz-plane and given the y coordinate, you need to create a slice in the zx-plane.

In each case, the index for the given axis is given by the second parameter `uint32_t slice`. You need to loop over the other two dimensions to copy over the voxel values.

e) Implement the out-stream appender function for `VoxelSlice`:

```
std::ostream& operator<<(std::ostream& ostream, const VoxelSlice& slice);
```

Output the voxels row by row. For each set voxel, output `'X'`, otherwise output `'.'`. After each voxel (including the last in the row), output a space `' '`. At the end of each row, output a new line `'\n'`.

When implemented correctly, appending a voxel slice to `std::cout` should print the voxel slice's contents to the console. The entire voxel grid can be printed by using its corresponding function

```
std::ostream& operator<<(std::ostream& ostream, const VoxelGrid& vg);
```

which creates slices in the z dimension and appends them to the out-stream.

f) Implement

```
bool isInside_impl(const Point3D &p) const override;
```

Check whether the voxel containing the 3D point is set. Account for the voxel grid's bounding box and the voxel grid's resolution. Return `false` for all points outside the bounding box. Use the `isSet` helper function to access the voxel's value and make sure to not access invalid elements when accessing the value for points which are equal to the maximum position of the bounding box in any dimension. The bounding box's `contains` function will consider such points to be inside.

This function is needed if one wishes to perform further operations on explicit voxel grids.

6.5 Construct Some Geometry (Bonus 10 Points)

C++

We encourage you to construct your own composite shapes. In `your_shape.cpp`, implement the `Shape your_shape()` function and create some nice geometry or patterns. Due to the large number of participants, we might not have the time to look at each submission in much detail but we plan to show off some of the nicer shapes and patterns you can come up with.

There are already some examples implemented for you to look at. To enable them, uncomment the line `target_compile_definitions(submission PUBLIC ENABLE_DEMO_CODE)` in the `CMakeLists.txt`.