Drawing shapes with ShapeContainer

Global settings

The VAirDraw base `Application` class stores and manages the applications's current drawing mode (lines or shapes). It is stored inside the `Application::drawingMode` attribute and must be set to `DrawingModes::LINES_AND_SURFACES` value in order to start drawing shapes.

This is what is done through the `Toolbox::CHANGE_DRAWING_MODE` action, associated with the drawing mode integer value as parameter. The example below shows the drawing mode configuration part, setting the initial mode to 0 (`DrawingModes::LINES_ONLY`).

```
"toolbox" : {
    "actions" : [
        [ <button_index>, "change-drawing-mode", 0 ]
    ]
}
```

Note that the mode value can also be set in `Toolbox::update()` method if you only need to switch between two modes with the same toolbox button.

```
ToolBox::Event &ToolBox::update(bool isDrawing)
{
    ...
    case CHANGE_DRAWING_MODE:
        {
            ...

            drawingMode = (drawingMode == LINES_ONLY) ? LINES_AND_SURFACES : LINES_O
            action.param = (int)drawingMode;
        }
        break;
    ...
}
```

Setting `drawingMode` to `LINES_AND_SURFACES` indicates the `Pencil` object to send input points to the `ShapeContainer` object used for drawing shapes in the scene. Each time `Pencil::update()` is called with new points to draw, the `Pencil` is given the right `LinesContainer` object to store them thanks to `Application::getDrawingContainer()` . The `Pencil` object also notifies the `ShapeContainer` object that new points have been provided to this `LinesContainer` if it belongs to a shape.

```
void Pencil::update()
{
    ...
    // when we detect that the user has provided new points
    if (app->getDrawingMode() == Application::LINES_AND_SURFACES) {
        notifyCount++;
        if (notifyCount >= NOTIFY_LIMIT) {
            app->getShapeContainer()->updateLastShape();
            notifyCount = 0;
        }
    }
    ...
}
```

`Pencil::NOTIFY_LIMIT` indicates the number of `Pencil::update()` calls before updating the resulting shape.

ShapeContainer

Every operation related to drawing and displaying shapes in the scene are handled by the `ShapeContainer` class (see `Apps/ShapeContainer.h`).

Container instanciation

`ShapeContainer` overrides the base class `Container` , so you just need to create and add a `ShapeContainer` object at the desired location inside the scene graph. Adding the object to `Application::mainScene.world` should be convenient for most usages.

```
// Within Apps/Application.cpp
shapeContainer = new ShapeContainer();
mainScene.world.addNode(shapeContainer);
```

Create and display shapes

A `ShapeContainer` object stores a list of `Shape` objects, which are used to monitor each shape's geometry computations and conversion into displayable meshes.

```
class ShapeContainer
{
    ...
    private:
        std::vector<Shape *> shapes;
    ...
```

```
}
```

A call to `ShapeContainer::beginShape()` creates a new `Shape`, which has its own `LinesContainer` reference. This is the object the `Shape` will look at when needing to retrieve the input points to generate a mesh. When the user is drawing, the `Pencil` asks the `Application` for the `LinesContainer` of the current shape (`ShapeContainer::getLastLinesContainer()`) in order to add new points to it.

Each call to `ShapeContainer::updateLastShape()` triggers the computation of a new mesh from the whole set of points stored inside the last `Shape`'s `LinesContainer` object.

The user also needs to end the current shape and start a new one. This is what `ShapeContainer::endLastShape()` stands for. It triggers the generation of a high resolution mesh, unlike "preview" meshes displayed while the user is still drawing which have lower resolution for performance.

If you just need to draw shapes one after another with no access to the previous ones, `beginShape()`, `getLastLinesContainer()`, `updateLastShape()` and `endLastShape()` should be sufficient. You can also access a specific shape, whatever it is the last or not, by specifying a unique ID as the first argument of the `beginShape()` method as shown below.

```
shapeContainer->beginShape(<shape_id>);
```

The shape's unique ID can be given as the first argument of `getLinesContainer()` and `updateShape()` to manipulate this shape using the same principle.

```
LinesContainer *lc = shapeContainer->getLinesContainer(<shape_id>);
// do stuff on the LinesContainer (add points, new lines...)
shapeContainer->updateShape(<shape_id>);

// ...
shapeContainer->endShape(<shape_id>);
```

This is precisely what is used when we retrieve shapes from files and want to generate them again inside the scene (see | `Apps/Replay/ReplayCapture.cpp` and `Apps/DrawCaptureLoader.cpp`).

Shape class

`Shape` encapsulates all the necessary features to compute and display mesh-based shapes from a `LinesContainer` input. For this purpose, it holds a reference to a `ShapeGeometry` object (`Shape::geometry`) which is provided with all the points from the `LinesContainer` each time `Shape::update()` is called.

```
bool Shape::update(const bool &highResolution, const glm::u8vec4 &color, const bool
{
    ...
    newLine = linesContainer->getLineNewVertices(lineNewVertices);
    geometry->addCurveVertices(lineNewVertices, newLine);
    ...

    ...
    geometry->updateSurface(highResolution);
    ...
}
```

The resulting mesh is then passed to a `ShapeView` object reference (`Shape::view`), which converts the result into a displayable mesh.

```
    view->update(geometry, color);
```

Computations with ShapeGeometry

`ShapeGeometry` holds and monitors all processes related to the computation of meshes from 3D points. It holds a reference to a `ApproxSurface` object responsible of the actual computation, based on a `CurvesSet` object storing the input points in the adapted format (see `SurfaceGenerator/ApproxSurface.cpp`).

It also manages the type of shape to be computed (`ShapeGeometry::type`). From the `Application` 's point of view, the type is modified using `ShapeContainer::setCurrentType(<type_value>` and changes are taken into account in the last `ShapeGeometry` object of the last shape.

The current shape's resolution values (low when drawing, high for ended shapes) are stored as static members of this class and can be set from the configuration file.

```
"application" : {
    ...
    "shapes" : {
        "resolution" : {
            "plane" : [30, 90],
            "octahedron" : [10, 30]
        },
    }
    ...
}
```

Display with ShapeView

As geometry is processed in a specific format (Eigen library data structures), resulting shapes need to be converted to displayable meshes. This is what `ShapeView` stands for. It retrieves the geometric result (vertices, faces and normales) as `Eigen` matrices and renders a `Mesh<MCVertex>`. It is responsible of its own rendering, overriding the base class `Node`. When a new shape is created from the `ShapeContainer`, the shape's view is added to this container along with the `LinesContainer` object to display the lines.

```
void ShapeContainer::beginShape(const int &shapeId, const int &userId, const int &ty
{
    // check if shape already exists
    ...
    // Create a new shape if necessary
    if (!shape) {
        shape = new Shape(shapeId, userId, type, sublayer);
        shapes.push_back(shape);
        this->addNode(shape->getLinesContainer());
        this->addNode(shape->getView());
    }
}
```

Every display option is handled by this class to generate the appropriate mesh. The name of shader programs used for rendering are stored as static members of the class (`ShapeView::shapeShaderPrograms`), mapped to each geometry type value. Shader programs can be set for each geometry type inside the json configuration file.

```
"application" : {
    ...
    "shapes" : {
        "shading-programs" : {
            "plane": "DiffuseMesh",
            "octahedron" : "ReflectMesh"
        }
    }
}
```

Other options are responsible of alternate and wireframe renderings (resp. highlight a specific shape when picked by the user, and display edges of the mesh). See `ShapeView::render()` for more details about how these parameters are handled when rendering the mesh.