Intersection detection with Collider

Presentation

The `Collider` class is a generic interface to find intersections with 3D objects in the VAirDraw scene, from either a *ray* or a *position*.

Each inherited class represents a shape-specific collision detection system. Some are implemented for simple shapes (`SphereCollider` , `CylinderCollider` , `BoxCollider` , `MeshCollider`). See `SceneGraph/Colliders/Collider.h` for more details.

A specific abstract class is used for mesh-based colliders (`MeshCollider`), as different *acceleration structures* exist to detect collisions on meshes.

Colliders are compatible with `model` and `3d-shape` content types in *scenery* files. The example below is available inside `assets/sceneries/test_collisions.json` .

```
{
"content" : [
    {
        "type" : "model",
        "file" : "models/batman/batman70.fbx",
        "transform" : { "translation" : [ 0.0, 1.0, 0.0 ] },
        "collider" : {
            "enable" : true,
            "debug-display" : true,
            "use-cache" : true,
            "closed" : true,
            "type" : "octree",
            "depth" : 5,
            "build-type" : "full",
            "proximity-search" : true
        }
    },
    {
        "type" : "3d-shape",
        "name" : "sphere",
        "dimensions" : { "radius" : 0.5 },
        "transform" : { "translation" : [ 0.0, 1.0, -1.0 ] },
        "collider" : {  "enable" : true }
    },
    {
        "type" : "3d-shape",
        "name" : "cylinder",
        "dimensions" : { "radius" : 0.2, "height" : 0.5 },
        "transform" : { "translation" : [ 1.0, 1.0, -1.0 ] },
        "collider" : {  "enable" : true }
    }
]
}
```

You can see that collider properties differ according to the content type. However all colliders provide at least `enable` and `debug-display` options. Note that the latter is implicitly enabled on `3d-shape` content type, as basic objects and their collider have the exact same geometry.

```
"collider" : {
    "enable" : true,
    "debug-display" : true,

    // ...
}
```

Mesh-based colliders specific properties

```
"collider" : {
    // ...

    "use-cache" : true,
    "closed" : true,
    "type" : "octree",

    // ...
}
```

Mesh-based colliders are already used on 3D models (`"type" : "model"`) and `Shape` objects stored inside `ShapeContainer` objects. For these, several options are available.

- `use-cache` : whether computed collider(s) must be stored in cache files. This highly increases the time for loading at next application startups. When adding the same model several times to the scene, consider using the **exact same collider configuration**. Otherwise, cache will work only for one of these configurations.
- `closed` : whether the underlying mesh has to be considered as closed. **Collision tests for open and closed surfaces are not the same**.
- `type` : type of acceleration structure to use for the collider. Two types have been available so far (`"bounding-box"`, `"octree"`). The first one is a simple bounding box. The second one's principle is detailed in the next section.

Octree colliders specific properties

```
"collider" : {
    // ...

    "depth" : 5,
    "build-type" : "full",
    "proximity-search" : true
}
```

When `type` is set to `octree`, a *tree structure* containing *cuboids* (i.e. sub-cubes of an initial bounding cube) is constructed (see https://en.wikipedia.org/wiki/Octree). Each cuboid potentially contains zero, one or several primitives from its proxy, referenced by an object ID and an element ID (see `IPrimitiveProxy::Element` in `Renderer/PrimitiveProxy.h`).

The octree traversal procedure via *raycasting* guarantees that we always retrieve, if existing, the **closest intersection** from the ray's origin.

- `depth` : number of *subdivision levels* inside the octree. The more you increase this number, the more mesh primitives are partitioned and intersection tests are fast. Note that performance stabilizes at depth levels around 6-7 where primitives dimensions become bigger than partitioning sub-cubes, making further partitioning useless.
- `build-type` : octree construction can use two different techniques
  - `"fast"` : primitives are stored in their smallest bounding node in the tree, **even if it is not a terminal one**
  - `"full"` : primitives are **only stored inside terminal nodes**. Note that triangles can sometimes **appear in several nodes**, as the only condition of storage of a primitive in a terminal node is intersecting with this node. As names suggest,

this method takes more (**and sometimes very long, be careful**) time to build the octree.

- `proximity-search` : whether the collider can use the *latest intersected node* when a new collision test is requested. When enabled, we proceed with an *down-to-up search* in the tree, starting from the last intersected nodes. This is particularly efficient when collider is used in a *continuous* intersection use case, i.e. where two successive intersections (in time) are often close to each other in space. This is particularly the case when drawing on a mesh, for example and can make the collider more efficient. This mode is only available with `build-type` set to `full`. Note that this traversal method **breaks the closest intersection guarantee**.

Usage from nodes of the scene graph

Collisions are tested on `Node` objects through the `Node::searchIntersection` method, with either a 3D position or a ray. The usual process is to traverse the scene graph from the scene root `Container` and test each composing `Node` children recursively. The example below illustrates a raycast initiated from the user's pointed direction.

```
// Apps/Pointer/Pencil.cpp
void Pencil::update()
{
  // ...

  Intersections::Ray rayToCast {glm::vec4(pointerPosition, 1.0f),
glm::vec4(pointerDirection, 0.0f)};
  IntersectionInfo *newIntersection = new IntersectionInfo();
  app->getWorldContainer()->searchIntersection(rayToCast, newIntersection);

  // ...
}
```

`IntersectionInfo` is the base class to store data about the collision if one was found. More information is added in some `Container` derived classes by overriding `IntersectionInfo` (see `Apps/LinesContainer.h`, `Apps/ShapeContainer.h`).

The `IntersectionInfo` object is instanciated from the caller, and the intersecting `Node`, if existing, fills the object with relevant data in its own `Node::searchIntersection()` method.

```
// Misc/Intersections.h
class IntersectionInfo
{
public:
    float distance {FLT_MAX};
    Node *node {nullptr};
    glm::vec3 position {0.0f};
    glm::vec3 normal {0.0f};

    IntersectionInfo() {}
    virtual ~IntersectionInfo() {}
    virtual void reset(float dist = std::numeric_limits<float>::max()) {
        distance = dist; node = nullptr; position = glm::vec3(0.0f); normal =
glm::vec3(0.0f);
    }
};
```

Collision detection inside a Node-derived class

Each `Node` has a `shared_ptr<Collider>` reference. The purpose of `Node::enableCollider` is precisely to create the desired `Collider` object according to the current `Node` derived class type. You can either instanciate *basic colliders* (see `SceneGraph/Colliders/BasicColliders.h`) or *mesh-based colliders* through the factory method `MeshCollider::create` (see `SceneGraph/Colliders/Collider.h`). An example of `MeshCollider` instanciation from `Node::enableCollider` overridden method is shown below in the `Mesh<VertexT>` class.

```
// SceneGraph/3DObjects/Mesh.inl
template<typename VertexT>
void Mesh<VertexT>::enableCollider()
{
    if (!colliderProperties || !colliderProperties->enabled) {
        logstreams::error << "Collider is disabled on mesh " << this <<
std::endl;
        return;
    }
    // TODO : change collider if a new one is requested...
    if (Node::collider) return;
    if (colliderPath.empty()) {
        logstreams::error << "No collider file name was specified on mesh " <<
this << ". Collider will not be enabled." << std::endl;
        return;
    }

    Node::collider = std::shared_ptr<Collider>(MeshCollider::create(this,
colliderProperties, colliderPath));
}
```

In this case, `Collider::Properties` are either loaded using a json-based object (see `collider` fields of `assets/sceneries/test_collisions.json`) through a factory method `Collider::importProperties`, or instanciated programmatically according to the `Collider` type you wish to use (see `OctreeCollider::Properties` in `SceneGraph/Colliders/OctreeCollider.h` for instance).

When the object has a reference to a `Collider` other than `nullptr`, `Node::searchIntersection` asks the `Node`'s `Collider` object for a potential collision with the point/ray. As every collider is built in the `Node`'s **local space**, the intersecting position/ray is always converted in this space before intersecting it.

```
// SceneGraph/Node.cpp
bool Node::searchIntersection(const Intersections::Ray &ray, IntersectionInfo
*&outResult)
{
    if (!collider || !isVisible()) return false;

    Intersections::Ray localRay = Intersections::RayApplyMatrix(ray,
getInvMatrix());

    if (collider->raycast(localRay, outResult)) {
        // If no additional information is needed by the caller
        if (!outResult) return true;

        outResult->position = glm::vec3(getMatrix() * glm::vec4(outResult-
>position, 1.0f));
        glm::mat3 normalMat =
glm::mat3(glm::inverse(glm::transpose(getMatrix())));
        outResult->normal = glm::normalize(normalMat * outResult->normal);
        outResult->node = this;
```

```
            return true;
        }

        return false;
}
```

The `IntersectionInfo` pointer is passed from the `Node` object to its `Collider` which provides geometric information about the intersection (distance, position and normal). The coordinates are then converted back into the **global space**.

You can override the generic `Node::searchIntersection` method if your `Node` derived class needs a different behaviour to detect intersections. For example, it would not make sense that the `Container` class have its own `Collider`. Colliders are yet useful for each of its children nodes. Therefore `Container::searchIntersection` just propagates the `searchIntersection` request to its children.

```
bool Container::searchIntersection(const glm::vec3 &position, IntersectionInfo
*&outResult)
{
    if (!isVisible()) return false;

    glm::vec3 localPosition = glm::vec3(getInvMatrix() * glm::vec4(position,
1.0f));
    for (const auto& node : nodes) {
        if (node->searchIntersection(localPosition, outResult))
            return true;
    }

    return false;
}
```

Implement a specific Collider

Base Collider class

Every `Collider` stores the necessary geometric information for its inner structure. Once constructed, the `Collider` must be able to test intersections with a local `Intersections::Ray` and a `glm::vec3` 3D position. For this purpose, implement `Collider::raycast` and `Collider::contains` for your `Collider` type. These implementations will then be called from the corresponding `Node`'s `searchIntersection` method when looking for intersections.

```
class Collider
{
public:
    // ...

    virtual bool raycast(const Intersections::Ray &ray, IntersectionInfo
*outResult = {nullptr}) { return false; }
    virtual bool contains(const glm::vec3 &point, IntersectionInfo *outResult =
{nullptr}) { return false; }

    // ...
}
```

You can also override the `Collider::debugDisplay` method to add a mesh representation of the collider geometry when debugging. See example below for a `BoxCollider`.

```
// SceneGraph/Colliders/Collider.cpp
void BoxCollider::debugDisplay(Container *container)
{
    Box *box = new Box(0.5f * (bbox.minPos + bbox.maxPos), bbox.size,
debugColor);
    container->addNode(box);
}
```

The `Intersections` namespace contains utility functions to test and compute intersections between basic 3D objects (see `Misc/Intersections.h`).

Mesh-based Collider classes

Some colliders need an access to their referring primitive's inner geometry to be constructed and be able to test intersections on them. *Proxies* (see `Renderer/PrimitiveProxy.h`) are used to protect geometric primitives while allowing this access.

If you want to implement a `Collider` based on a triangular mesh structure, use the `MeshCollider` base class. It gives an access to the associated mesh's geometry through the `IPrimitiveProxy` interface, which can be used to build the collider's inner structure.

```
// SceneGraph/Colliders/OctreeCollider.h
class MeshCollider : public Collider
{
// ...

public:
    static std::shared_ptr<MeshCollider> create(IPrimitiveProxy *px,
std::shared_ptr<Properties> props, const std::string &cacheFilePath, const
unsigned int &importFlags = {});

    MeshCollider(IPrimitiveProxy *meshProxy) : meshProxy(meshProxy) { }

    // ...
};
```

The usual process is to implement the `PrimitiveProxy<PrimitiveT, EltS>` interface using *CRTP* (see https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern) for your primitive, and then creating your collider from this primitive *inner scope*, specifying `this` as first parameter the to `MeshCollider` constructor (see instruction from `Mesh<VertexT>::enableCollider()` below).

```
// SceneGraph/3DObjects/Mesh.inl
template<typename VertexT>
void Mesh<VertexT>::enableCollider()
{
    // ...

    Node::collider = std::shared_ptr<Collider>(MeshCollider::create(this,
colliderProperties, colliderPath));
}
```

Along with configuration options from `MeshCollider::Properties` explained before, you can also *disable the use of proxies* on a collider by calling `MeshCollider::setUseProxy(false)`. In this case, intersections will **only be computed on the collider base structure** (terminal sub-cubes for octrees) instead of primitive elements such as triangles, thus giving **more approximate positions and orientations** but should result in faster intersection tests.

Overridden methods

Any mesh-based collider must override some methods to function properly.

```
virtual void compute() { }
```

Place inside `compute`'s body all the required instructions to build your `Collider` structure.

```
virtual std::string getFileExtension() { return ""; }
```

Return the extension string to set on cache files of this specific collider (".octree" for octree colliders for example).

```
virtual bool read(std::ifstream &colliderInput) { return false; }
```

Use the provided input file stream (previously initialized) to build your collider structure according to your *own storage format*. Boolean return value should be `false` when something went wrong during the read operation, and `true` otherwise.

```
virtual bool write(std::ofstream &colliderOutput) { return false; }
```

Use the provided output file stream (previously initialized) to store your collider structure, following of course the *same format* as `read()` if you want your caching system to work.

You can always check `SceneGraph/Colliders/OctreeCollider.cpp` for an example of implementation of these methods.