Exploring future C++ features within a geometric modeling context

Jostein Bratlie and Rune Dalmo

UiT - the Arctic University of Norway

October 20, 2019

Abstract

The development of the C++ programming language and its standard library has undergone a renaissance since the emerge of the C++11 standard. Through modern features, expansions to the standard library and simplifications the language has become more relevant than ever before. Comparing past and future feature sets (C++17, C++20, ...) is somewhat similar to comparing different programming languages. In this article we address how new and upcoming features of the language can be utilized to ease the development of domain specific application areas through features such as *non-intrusive inheritance*, *semantic compile-time polymorphism* and type safety. We provide representative examples by application to differential geometry by modeling a hierarchical structure for parametric object evaluation.

1 Introduction

Differential geometry is a field within mathematics where the theory of differential manifolds [2], such as curves and surfaces, and Riemannian spaces [3] are central. The R&D group Simulations at UiT Narvik has conducted research within this field since the mid 1990s, with a special focus on modeling techniques of spatial objects and spline theory. Realized implementations of these mathematical and geometric concepts and constructions are considered as important tools for visual understanding and verification throughout the R&D work process. As a result, an in-house software library named GMlib [8], which aids in this aspect of the R&D work, has emerged over the years. GMlib is a fairly clean C++ library with only a few external dependencies besides the standard template library (STL). However, due to its age and maturity, the code base, written in C++98, consists of major parts of legacy code which now faces a set of challenges in the future. After an internal review in 2017 it was decided to construct a new prototype where the focus should be on statically descriptive tools for the differential geometric modeling aspects of our R&D work. In addition to historical reasons for continuing to use C++ we are fond of its closeness to hardware, optimization potential, no-overhead principles,

This paper was presented at the NIK-2019 conference; see http://www.nik.no/.

and the support for generating bindings to other programming languages, such as, for example, Python. The major focus points for the development process are the following:

- light-weight header-only library which follows the C++ core language guidelines [7] and targets the latest C++ standard (current: C++17 [6]),
- restrict library maintenance to key R&D activities, and
- optimize for rapid prototyping of domain-specific features and strong types.

After an analysis we decided that the dependency needs of the new code base was limited to three C++ libraries, besides the STL; namely, Blaze [4] for basic linear algebra, OpenMesh [1] for triangular- and polygonal meshing and topology, and Qt [12] for matters related to graphics, application framework and demo control. The libraries were selected due to their use of modern C++ features, which we depend on, their proven stability, and their ability to target a variety of architectures; from micro-controllers to smartphones and desktop- and backend systems. Both the old and new prototype libraries and demo applications can be found at the Nordic e-Infrastructure Collaboration (NeIC) software hub [14], under the project "gmlib".

2 Problem setting

The intended use of the new software library is as a tool to verify, very strictly, abstract mathematical concepts and constructions, where compile-time rather than runtime-rules are utilized to catch constructional errors. A way to realize this is by using strongly typed expressions. In C++ this translates into a subset of features which enhances the static type system.

This article focuses on certain new and future language features and how they can be utilized to enforce our R&D work. The article introduces a set of design techniques and discusses how they are beneficial in programming of geometric modeling concepts. The techniques are exemplified via building a small application programming interface (API) throughout the article. This API is used to model embedded spatial-hierarchical parameterized objects. Examples of such constructions can be

- a sphere, torus or Bézier surface, which all are two-dimensional objects that can be embedded in a three-dimensional space,
- or a multi-hierarchical object chain, such as the ones illustrated in fig. 1,

of which the latter example is illustrated in the left part of fig. 1. Such problems, where the concatenated and multi-hierarchical are relevant scenarios, quickly become complex and challenging to model programmatically with confidence.

Implementation wise, if considering these objects as part of an infinite set, the natural modeling technique used in C++ is by utilization of abstract interfaces and virtual functions. This in turn leads to the use of techniques which builds on dynamic casting and runtime type information (RTTI). On the other hand, by considering these hierarchical objects as parts of a finite set of objects we can use a different set of C++ modeling techniques and utilize much more of the static type system, which again alleviates the use of runtime-centric techniques.

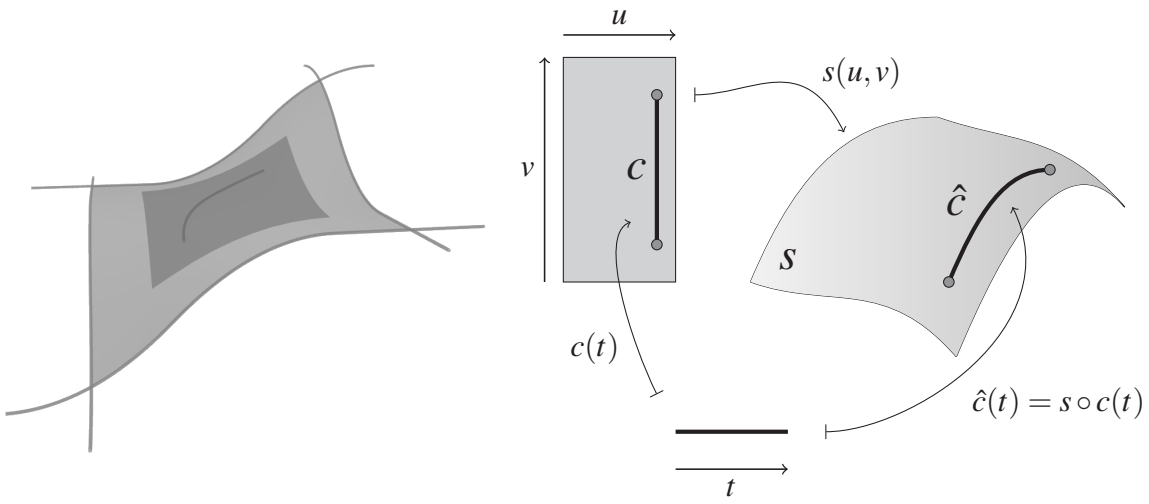

Figure 1: Left: a rendering of inheritance-hierarchical parametric objects; On the boundary we have b-spline curves ($\mathbb{R} \mapsto \mathbb{R}^3$), from these we have sub-curves ($\mathbb{R} \mapsto \mathbb{R}$) which again are input to a bi-linear Coons patch surface, ($\mathbb{R}^2 \mapsto \mathbb{R}^3$). From this Coons patch we have a sub-surface, plane ($\mathbb{R}^2 \mapsto \mathbb{R}^2$), and finally from this we have a sub-curve, 5^{th}-degree Hermite curve ($\mathbb{R} \mapsto \mathbb{R}^2 \mapsto \mathbb{R}^3$). Right: a mathematical sketch of spatial mappings: $\hat{c}(t) = s \circ c(t)$

The article is divided into three sections; *principal design techniques*, an exemplified *parametric object API* and a section which discusses pros and cons of these techniques and explores new and future language features. The considered new and future language features primarily enhances these principal design techniques as well as alleviates boiler plate code and therefore reduces the complexity and amount of programmatic errors.

3 Principal design techniques

C++ is a mature well behaving language possessing a set of features which are desirable for system design. One of these features is its well defined type system, which consists of a finite set of built-in types and a single user-defined type: `class` /`struct`. The user defined type is used to describe how objects are laid out in memory. Its associated member functions describe an object's semantics and related operations. As the language only consists of a single user-defined type, the inheritance model is equally well defined, and sub-objects with altered semantics are handled by the use of virtual tables. Furthermore, this yields a memory model that reflects the system memory for a target architecture and the requirement that all types are known at the point of compilation.

Due to its memory model, C++ supports casting between dynamic objects via runtime mechanisms. Dynamic casting is a powerful feature, but it comes with a possible high cost with respect to resources. The complexity of this and related mechanisms depend on the relationships between the dynamic objects and the hierarchical class structure. Additionally, that introduces a weak object model since an object's inheritance from a base type to a deduced type is subject to runtime checks. Nevertheless, this is the de facto method used to design object oriented problems, where the possible objects from a given class hierarchy represents an infinite set where there is a parent-child relationship.

We propose to take an alternative approach to such design problems where we consider our infinite set of potential sub-classes to be finite at compile time. This allows us to use other mechanisms besides virtual tables and for this purpose disable the RTTI. In the following sections this is exemplified through a small proposed API. We start by discussing a set of key C++ implementation techniques used to realize the static flavor of polymorphism, namely,

- non-intrusive inheritance,
- semantic compile-time polymorphism, and
- aggregated properties and transitive constructors.

Non-intrusive inheritance

The term non-intrusive inheritance refers to an inheritance model where a derived class can inherit a realized base class, possibly unknown at the time of design, where the realized base class adheres to a set of static rules such that the memory model is preserved at the time of compilation. For instance, the derived class is defined in a library, but the realized base class can be designed by the library's end-user. To facilitate non-intrusive inheritance one can utilize variadic class templates; `template <typename... Ts>`, which accepts one or more template parameters into a template pack. The ellipsis operator, `...`, is then used to unpack the template arguments:

```
struct SceneObject {}; struct Base {};

template <typename... Base_Ts> struct Object : Base_Ts... {};

using NonInheritingObject = Object<>;
using ObjectOfBase        = Object<Base>;
using SceneObjectOfBase   = Object<SceneObject,Base>;
```

Semantic compile-time polymorphism

The standard mechanisms used in C++ to implement polymorphism are function overloading and virtual functions. The latter enables runtime-dynamic types where the semantics of a derived object can be reimplemented for a given virtual function in a derived class and then be called for the derived object dynamically through a pointer of the base class type. By flattening the structure, using a middle layer *kernel* type, utilizing *non-intrusive inheritance*, and exploiting function overloading, we can mimic much of the same behaviour for semantic compile-time polymorphism. Consider the following type definitions:

```
                    struct Base {};
<typename Kernel_T> struct Object : Kernel_T {};

template <Base_T> struct KernelA : Base_T {
  auto evaluate( int par ) { return Vector{par}; }
  auto evaluate() { return double(0.5); } };

template <Base_T> struct KernelB : Base_T {
  auto evaluate() { return double(0.5); } };

template <typename Obj_T, typename... Params_Ts>
auto evaluate(Obj_T t, Param_Ts... params){
  return t.evaluate(params...); }
```

The `Object` class inherits a potential kernel which again inherits a common base.

```
using ObjectA = Object<KernelA<Base>>;
using ObjectB = Object<KernelB<Base>>;
```

The defined objects `ObjectA` and `ObjectB` share the same polymorphic API, where the return type is deduced by binding to the unconstrained placeholder type `auto`. Both object types can then be utilized in polymorphic contexts, as follows:

```
auto A = ObjectA; auto B = ObjectB;
auto a = evaluate(A,4); // Ok
auto b = evaluate(A);   // Ok
int  c = evaluate(A,4); // Compile error:    return type mismatch
auto d = evaluate(B,4); // Compile error: parameter type mismatch
```

Aggregated properties

Static properties can be utilized to hold compile-time types or sizes such as an object's spatial dimension or base unit type. These are properties which are changed for instance to increase computational precision (e.g. *long double* or *integer* over *floats*), or to specify the dimension of an embedding space (e.g. two-dimensional, $\mathbb{R}^2$, instead of three-dimensional, $\mathbb{R}^3$). Such internal properties can be defined using static constant expressions or `using` type definitions as part of *non-intrusive inheritance* (is-a relationship), or as internal aggregated type definitions (is-a or has-a relationship). Static properties are not part of an instantiated object by definition, but can be stored if needed. As such, this mechanism is powerful and inflicts no run-time penalty. However, such properties are not aggregated through inheritance and must therefore be explicitly defined, as follows:

```
struct StaticProperties {
  using                Unit      = int;
  static constexpr auto Dimension = 3ul; };

template <typename Props_T> struct Object {
  using                Unit      = typename Props_T::Unit;
  static constexpr auto Dimension = Props_T::Dimension; };
```

Transitive constructor inheritance

When utilizing a *non-intrusive inheritance model* we need a mechanism to transitively aggregate the future constructors of a class-type we do no yet know exists. This can be achieved using a combination of *templated constructors*, *variadic templates* and *perfect forwarding*. Let us exemplify it through an internal sub-object construction with a has-a relationship, as such:

```
template <typename SubObj_T> struct Object {
  SubObj_T  sub;
  template <typename... Ts> Object(Ts&&... ts)
    : sub(std::forward<Ts>(ts)...){} };
```

Given an object type `A` for an internal sub-object with the following constructors

```
struct A {
  explicit A(float f) {}
  explicit A(std::array<int,3> i) {} };
```

yields the following natural, but implicit aggregate, syntax:

```
auto obj_one Object<A>(2.3f);
auto obj_two Object<A>({1,2,3});
```

Simplifying API through type deduction

Finally, it is useful to simplify an API without introducing new types. This can be achieved by using type definition via the `using` directive. Contrary to the pre-C++11 standard's `typedef` directive, `using` directives can be templated. Consider this API for a parametric object

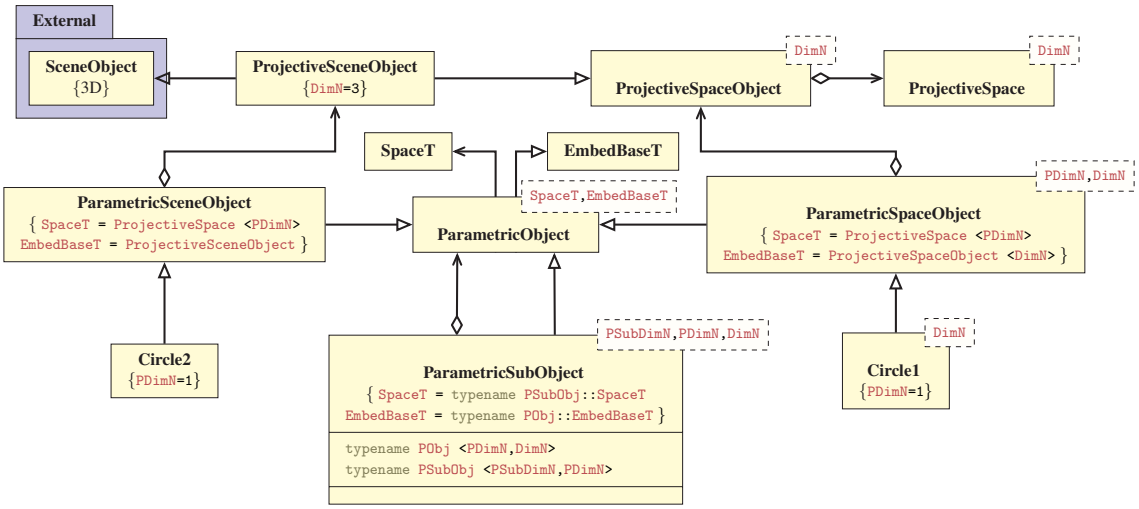

Figure 2: A templated model showing inheritance and semantic polymorphism.

```
template <size_t PSpaceDim, size_t EmbedDim> struct Object;
```

Specialized semantic APIs for curves and surfaces can then be constructed as follows:

```
template <size_t EmbedDim> using Curve   = Object<1ul,EmbedDim>;
template <size_t EmbedDim> using Surface = Object<2ul,EmbedDim>;
```

4 The parametric object API

Utilizing the above techniques, this section introduces a representative example API for the purpose of building parametric object spaces of the kind illustrated in fig. 1. A simplified API type diagram is shown in fig. 2.

Projective space

The projective space is common when working with visual parametric modeling. It is an affine mathematical space (see e.g. literature on differential- or Riemannian geometry [2, 3]), consisting of points and vectors with respect to a perspective projection such that parallel lines end up in a point at infinity, e.g., the horizon. The projective space meta-data description consists of its affine space containers for points and vectors in addition to a numerical unit data type and a space dimension. The space is usually represented as a frame in the shape of a homogeneous matrix. We model the meta-information of this mathematical space statically by using an *information only structure*:

```
template <typename Unit_T, size_t Dim_T> struct ProjectiveSpace {
  static constexpr auto Dim = Dim_T;    // Space dimension
  using Unit    = Unit_T;               // Storage unit type
  using Point   = Vector<Unit, Dim>;    // Affine space data types
  using Vector  = Vector<Unit, Dim>;
  using Frame   = Matrix<Unit, Dim>; };
```

From this, one can model a projective space object using a static *has-a* relationship.

```
template <typename EmbedSpace_T> struct ProjectiveSpaceObject {
  using EmbedSpace = EmbedSpace_T;          // Projective space
  using Unit = typename EmbedSpace::Unit; // Type aggregation
  /* ... */
  Frame frame = {/* identity */};           // Affine space data structure
  void translate( Vector ) { /*math*/ };  // Affine operations
  /* ... */ };
```

Parametric objects

A parametric object is a user defined type describing a differential mapping from an n-dimensional parameter space into a m-dimensional embed space, under certain restrictions. This can be modeled using the principles of *non-intrusive inheritance*, where a parametric object inherits a template kernel type, `Kernel_T`, realizing the actual parametric object.

```
template <typename Kernel_T>
struct ParametricObjectImpl : Kernel_T {
  // Context
  using Kernel = Kernel_T;
  // Embed Space
  using EmbedSpaceObj = typename Kernel::EmbedSpaceObj;
  using EmbedSpace    = typename Kernel::EmbedSpace;
  using Unit          = typename Kernel::Unit;
  /* ... */
  // Parametric space
  using PSpace        = typename Kernel::PSpace;
  using PSpaceUnit    = typename Kernel::PSpaceUnit;
  /* ... */
```

The specific space types are inherited from the kernel as aggregated properties and the member functions are inherited from the kernel according to the rules of inheritance. These can be used to construct chaining operations; such as transformations or evaluation with respect to a *parent* space (`frame`).

```
  auto evaluateParent(PSpacePoint par) {
    return frame * evaluate(par); }
  /* ... */
```

The kernel's constructor is inherited using *transitive constructor inheritance*, effectivly translating the kernel's constructor into the parametric object's own constructor. This can also be done for other *internal* kernel methods, such as private sub-methods.

```
  template <typename... Ts>
  explicit ParametricObjectImpl(Ts&&... ts)
    : Kernel(std::forward<Ts>(ts)...) { }
}; // END ParametricObjectImpl
```

A parametric object kernel

A specialized parametric object is then realized as a parametric object of a provided kernel. As an example, the parametric object kernel, `CircleKernel`, is constructed as a user defined type inheriting the projective space object as a template type, `EmbedSpaceObj_T`. Then as with the parametric object itself we use *aggregated properties* to inherit the static types. Additionally we define equivalent projective space types for the parameter space itself and extra utility types, prefixed `PSpace`. This is exemplified using a parameterized circle $C(t) = (x(t), y(t), z(t)) : \mathbb{R} \mapsto \mathbb{R}^3$.

```
template <typename  EmbedSpaceObj_T> struct CircleKernel
  : EmbedSpaceObj_T {
  using EmbedSpaceObj= EmbedSpaceObj_T;                  // Embed Space
  using EmbedSpace    = typename EmbedSpaceObj::EmbedSpace;
  /* ... */
  using PSpace        = space::ProjectiveSpace<1ul,Unit>; // PSpace
  using PSpaceUnit    = typename PSpace::Unit;
  static constexpr auto PSpaceDim = PSpace::Dim;
  /* ... */
  using PSpaceBoolArray = array<bool, PSpaceDim>;        // Utility
  using PSpaceSizeArray = array<size_t, PSpaceDim>;
```

The above code in fact represents boilerplate code for any parametric curve. The kernel-specifics for the parametric circle, such as data members, constructor and semantic polymorphic functions, can then be defined as follows:

```
  Unit r;                                          // Member: radius
  template <typename... Ts>                        // Transitive constructor
  CircleKernel(Unit radius = 3, Ts&&... ts)
    : Base(std::forward<Ts>(ts)...), r{radius} {}
                                                   // Semantic polymorphic functions
  auto            evaluate(PSpacePoint par) const {
     const auto& [t] = par;

     const auto x    = r * std::cos(t);
     const auto y    = r * std::sin(t);

     if      constexpr (Dim == 2) return Point{x, y};    // 2D
     else if constexpr (Dim == 3) return Point{x, y, 0}; // 3D
  }
  PSpaceBoolArray isClosed()       const { return {true}; }
  PSpacePoint     startParameter() const { return {0}; }
  PSpacePoint     endParameter()   const { return {2*M_PI}; }
}; // END CircleKernel
```

The template dependent *constexpr if* statement enables us to specify different return types for each supported embed space dimension.

Sub-space objects

The driving motivation to develop such an API has been to enable strong typing of hierarchical object structures such as the ones addressed in section 2. Contrary to the basic parametric object, a sub-object kernel is defined by a parametric object which is embedded in the parameter space of another parametric object. This can be achieved by defining the kernel and implementation of the sub-object slightly different to the parametric object type itself. However, as an invariant, the internal type-definitions should be kept identical to the ones of the parametric object, such that sub-object chains can be defined.

The kernel is defined from three types; the subbed parametric object, `ParametricObject_T`, a parametric object embedded in that parametric object's parametric space, `PSpaceObject_T`, and an embed space object, `EmbedSpaceObject_T`, which usually is the same as the embed space of the sub-object. This gives us a parametric object with three distinct spaces: the embed space of the parametric object, `EmbedSpace`, the parametric space of the sub-object, `PSpace`, and the parametric space of the parametric object, `ParametricObject_PSpace`.

```
template <typename PSpaceObject_T, typename ParametricObject_T,
          typename EmbedSpaceObject_T>
struct CurveInSurfaceKernel : EmbedSpaceObject_T {
  using PSpaceObject            = PSpaceObject_T;      // Context
  using ParametricObject        = ParametricObject_T;
  using EmbedSpaceObject        = EmbedSpaceObject_T;

  using EmbedSpace              = typename EmbedSpaceObject::EmbedSpace;
  using PSpace                  = typename PSpaceObject::PSpace;
  using ParametricObject_PSpace = typename ParametricObject::PSpace;

  using Unit                    = typename EmbedSpace::Unit;
  /* ... */
```

This leaves us with a similar boilerplate code as for the parametric circle kernel, but with an additional type set representing the common parametric space. Continuing, the sub-object's data members, constructor and polymorphic parametric object methods are defined as follows.

```
  PSpaceObject      pspace_object;      // Members
  ParametricObject* parametric_object;

  template <typename... Ts>             // Transitive constructor
  explicit SubCurveKernel(ParametricObject* obj, Ts&&... ts)
  : Base(), pspace_object(std::forward<Ts>(ts)...),
                          parametric_object{obj} {} };

  auto evaluate(PSpacePoint par) const  // Semantic polymorphic functions
  {
```

```
    const auto pspace_res = pspace_object.evaluate(par);
    const auto parametric_object_res =
      parametric_object->evaluate(pspace_res);
    return parametric_object_res;
  }
  auto isClosed() const { return pspace_object.isClosed(); }
  /* ... */
}; // END CurveInSurfaceKernel
```

Parametric sub-object implementation type

The implementation type for the parametric sub-object type is a small variation of the parametric object's type. Types are aggregated from the sub-object-in-object kernel which transitively calls the right constructor combination. Again, it is important that it retains the same type definition interface as the `ParametricObjectImpl` such that it also can be used in sub-object chain constructions.

```
template <typename Kernel_T> struct ParametricSubObjectImpl
  : ParametricObjectImpl<Kernel_T> {
  // Context
  using Kernel           = Kernel_T;
  using PSpaceObject     = typename Kernel::PSpaceObject;
  using ParametricObject = typename Kernel::ParametricObject;
  using EmbedSpaceObject = typename Kernel::EmbedSpaceObject;
  // Aggregated types
  using PSpace      = typename Kernel::PSpace;
  using EmbedSpace = typename EmbedSpaceObject::EmbedSpace;
  using ParametricObject_PSpace =
    typename Kernel::ParametricObject_PSpace;
```

Object creation is handled through a transitive constructor, which can be enforced by deleting the sub-objects default constructor.

```
  ParametricSubObjectImpl() = delete; // Delete default constructor
  template <typename... Ts>          // Transitive Constructor
  explicit ParametricSubObjectImpl(ParametricObject* obj,
                                   Ts&&... ts)
    : Base(obj, std::forward<Ts>(ts)...) {}
}; // END ParametricSubObjectImpl
```

User space API

C++ template code naturally results in low readability API's, however, as discussed in section 3, this can be mended utilizing `using` type-definitions:

```
// ParametricObject
template <template <typename> typename Kernel_T,
          typename EmbedSpaceObject_T>
using ParametricObject
  = ParametricObjectImpl<Kernel_T<EmbedSpaceObject_T>>;

// ParametricSubObject
template <template <typename> typename PSpaceKernel_T,
          template <typename, typename, typename>
          typename Kernel_T,
          typename ParametricObject_T>
using ParametricSubObject =
  ParametricSubObjectImpl<Kernel_T<
    ParametricObjectImpl<PSpaceKernel_T<ProjectiveSpaceObject<
      typename ParametricObject_T::PSpace>>>,
    ParametricObject_T,
    ProjectiveSpaceObject<
      typename ParametricObject_T::EmbedSpace>>>;
```

which leads to the following API usage:

```
using ProjSpace    = ProjectiveSpace<double,3ul>;
using ProjSpaceObj = ProjectiveSpaceObject<ProjSpace>;
using Circle       = ParametricObject     <CircleKernel, ProjSpaceObj>;
using Torus        = ParametricObject     <TorusKernel,  ProjSpaceObj>;
using SubCircle    = ParametricSubObject  <CircleKernel, SubCurveKernel, Torus>;

auto circle          = Circle();
auto torus           = Torus();
auto torus_subcircle = SubCircle(&torus, 2.0);
```

5 Concluding remarks

Utilizing the principles of *non-intrusive inheritance* and *semantic compile time polymorphism* we can redefine the object space of our modeling problem from an infinite to a finite set of possible objects. This allows us to design our C++ programs using static compile time techniques, which in turn lets us, most notably; to *move viable logic from run-time to compile-time*, and to apply *stronger types*. Consequently, fewer potential runtime errors occur since they can be caught at compile time and we can dismiss entire type categories by definition. Furthermore, the paradigm enables us to apply static analysis tools to problems where we earlier had to perform dynamic analysis. This again reduces errors and potentially leads to faster-runtime code; for example by dismissing branching opportunities.

On the other hand template programming makes libraries more complex, which again puts more responsibility on library programmers to construct understandable APIs. One drawback of a templated codebase is increased compile times and compile time resources. If not handled with care the increase in compile time and resources can grow exponentially. However, this will be alleviated with the upcoming C++20 feature; *modules*. Modules will mitigate the compile time overhead endured from repeated recursive header includes as well as the header inclusion mangling itself. Another drawback, if not handled with care, is the combinatoric nature of templates. As all function candidates, types and branches must be known at compile time, all permutations of possible template combinations are computed and evaluated by the compiler. This can then make both compile times and size of executables grow exponentially.

6 Future work

Let us take a look at future language features which will further help the verification of our R&D work and ease devlopment of such hierarchical class structures. C++20 introduced a new set of features. This is the biggest jump in a standarized feature level since the emerge of C++11. In this section we argue what we see as the most prominent feature with respect to the current proposed API, namely *Concepts*. Furthermore we briefly touch on a set of future features which are in development and planned for inclusion in standardized C++ over the next decade.

Concepts

In today's codebase *concepts* [10] are represented by *type-traits* and enforced through the SFINAE-mechanism (substitution failure is not an error). SFINAE together with *type-traits* limits the type space a template variable can occupy, and is used to remove template function candidates, at compile time, if type deduction fails.

Concepts on the other hand builds on the placeholder type `auto`. The type `auto` is the value-equivalent of `typename`. It is deduced at compile time, upon definition. With

```
auto i = int{4};
```

the type of the variable `i` is deduced to integer, while in the following example the type is deduced to the return-type value of the factory function.

```
auto curve = circleFactory(3.0);
```

A concept type is a restricted placeholder type. Imagine an algorithm which finds the intersection between two parametric objects. In today's codebase a generic

intersect algorithm is represented as a templated function over two unconstrained typenames; returning some result:

```
template <typename ObjA_T, typename ObjB_T> auto intersect(ObjA_T, ObjB_T);
```

In theory, types `ObjA_T` and `ObjB_T` can take any type. The error checking is delayed to later in the code where the functionality and types are realized. This leads to obscure error messages often hidden inside a static template-recursion layer.

An intersect method written using concept mechanisms would look something like the following (using the work group's short-hand notation [5]):

```
template <> IntersectResult auto intersect(Curve auto, Surface auto);
```

where `Curve`, `Surface` and `IntersectResult` are concept placeholder types. In this scenario the compiler deduces the placeholder types at the time of binding and can give an error message based on the restricted concept types, for instance: "function parameter one is not a curve, because ...".

The improved error messages is an appreciated side effect, but the real benefit is that we can model concepts by their unique and shared properties rather then restricting their incidental side effects.

Reflection, meta classes and contracts

Reflection and metaclasses introduces compile time functionality, building on the C++20 feature `consteval`, to either reflect upon existing user defined source elements or inject program source fragments at *compile-time*. Using syntax rules from the proposal paper [11], we can define a parametric circle type as follows:

```
class(ParametricCurve) Circle {
  using Unit                = double;
  static constexpr auto Dim = 3ul;

  PSpacePoint     startParameter() const { /**/ };
  PSpacePoint     endParameter() const   { /**/ };
  PSpaceBoolArray isClosed() const       { /**/ }
  Point           evaluate(PSpacePoint par) const { /**/ } };
```

The feature then dictates that our user-defined type, `Circle`, needs to comply with the rules posted through the meta-class `ParametricCurve`. If so, a set of the code fragments, completing the parametric curve construction, will be statically generated. This does not inflict any implicit inheritance.

Contracts on the other hand is a run-time tooling feature which documents and enforces invariants. For instance in the parametric object construction we could apply the following value invariants (this example uses syntax from the proposal [9]).

```
Point evaluate(PSpacePoint t) const
  [[ expects:             t >= startParameters() and t <= endParameters() ]]
  [[ ensures default res: not std::isnan(res) ]]
{
  [[ assert: not std::isnan(t) ]]
}
```

The `expects` directive is applied to input parameters, the `ensures` directive can reflect constraints onto the return value, and the `assert` directive produces assertion conditions. Contracts provides a functionality to turn on restrictions during development.

7 Acknowledgments

The prototype library, GMlib2, is a work in progress; therefore the article is accompanied by a small source code base containing a proposed principal

implementation example of the discussed application programming interface (API), which can be obtained by contacting the authors. The authors acknowledge the equal contribution norm [13] associated with alphabetical listing of authors.

References

[1] Mario Botsch, Stefan Steinberg, Stefan Bischoff, and Leif Kobbelt. OpenMesh: A Generic and Efficient Polygon Mesh Data Structure. In *OpenSG Symposium 2002*, 2002.

[2] M.P. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, 1976.

[3] M.P. do Carmo. *Riemannian Geometry*. Mathematics (Boston, Mass.). Birkhäuser, 1992.

[4] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: A performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[5] ISO. *ISO/IEC JTC! SC22 WG21 N4549 — Programming languages — C++ Extensions for Concepts*. 2015.

[6] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth edition, December 2017.

[7] IsoCpp. C++ core guidelines. https://github.com/isocpp/CppCoreGuidelines, 2019.

[8] Arne Lakså, Børre Bang, and Arnt Roald Kristoffersen. GM_lib, a C++ library for geometric modeling. Technical report, Narvik University College, Narvik, Norway, 2006.

[9] G. Dos Reis, J. D. Garica, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. Technical Report p0542r5, 2018.

[10] Aleksander Stepanov and Paul McJones. *Elements of Programming*. Semigroup Press, authors'/second edition, 2019.

[11] Herb Sutter. Metaclass functions: Generative C++. Technical Report p0707r4, 2019.

[12] The Qt Company. Qt - complete software development framework. https://www.qt.io, 2019.

[13] Teja Tscharntke, Michael E. Hochberg, Tatyana A. Rand, Vincent H. Resh, and Jochen Krauss. Author sequence and credit for contributions in multiauthored publications. *PLoS biology*, 5(1):e18, 2007.

[14] UiT - The Arctic University of Norway. GMlib/GMlib2 C++ geometric modeling library. (NeIC) https://source.coderefinery.org/gmlib, 2019.