Use of serialization for resource caching in 3D framework

Problem description from github issue

Sometimes, the loading of resources can take a long time. This is an issue, especially when loading larger 3D models or ORCA scenes like the NVIDIA Emerald Square City Scene. Such a scene can already be loaded with the orca_loader example => by default, it loads a "Sponza" scene (which is included directly in this repository), but it offers a button in the UI to load an ORCA scene from file. Those ORCA scenes are too big to be included directly in the repository. Because they are so big, loading takes a long time.

Hint: If there is no .fscene file included in the "Emerald Square City" scene, please contact @alexc71.

Loading is slow because 1) ASSIMP is parsing the loaded 3D model files, 2) ASSIMP is potentially applying some transformations, 3) the application queries *and copies* all requested mesh information from ASSIMP, 4) the application loads all images from file, 5) the application potentially transforms the loaded images (e.g. flipping them).

All of these steps happen before further uploading the resources to the GPU and can take a lot of time. They could be tremendously accelerated by loading the already prepared data from **a binary cache file**. I.e. it would work as follows: When the application first runs, it has to perform all of the above steps. Then it would serialize the loaded data into binary cache files. At subsequent starts, the application could check whether or not such a binary cache file already exists, and if it does, deserialize the data from it.

Which data should be serializable/deserializable?

- std::tuple<std::vector<glm::vec3>, std::vector<uint32_t>> as returned by gvk::get_vertices_and_indices
- std::vector<glm::vec2> as returned by gvk::get_2d_texture_coordinates or gvk::get_2d_texture_coordinates_flipped
- std::vector<glm::vec3> as returned by gvk::get_normals, gvk::get_tangents, gvk::get_bitangents, or gvk::get_3d_texture_coordinates
- std::vector<glm::vec4> as returned by gvk::get_colors or gvk::get_bone_weights
- std::vector<glm::uvec4> as returned by gvk::get_bone_indices
- gvk::material_gpu_data and std::vector<gvk::material_gpu_data> as returned as the first tuple element from gvk::convert_for_gpu_usage
- "std::vector<avk::image_sampler>" as returned as the second tuple element from gvk::convert_for_gpu_usage. This one is a bit tricky (that's why it has been put into

quotes) => avk::image_sampler can not be serialized/deserialized directly because it represents Vulkan-handles which represent data on the GPU. The tricky part here is that actually the function gvk::convert_for_gpu_usage has to be extended, or an alternative "*_cached"-function has to be implemented which handles caching of images internally. Images are loaded within this function from file (see material_image_helpers.cpp#L231) or from memory (see material_image_helpers.cpp#L195 and material_image_helpers.cpp#L211). Optimally, the image data would also be cached. (Further details see above => "Serializing/deserializing image data")
- gvk::lightsource_gpu_data and std::vector<lightsource_gpu_data> as returned from gvk::convert_for_gpu_usage.
- gvk::animation as returned by model_t::prepare_animation_for_meshes_into_strided_contiguous_memory, model_t::prepare_animation_for_meshes_into_tightly_packed_contiguous_memory, or model_t::prepare_animation_for_meshes_with_offsets; and also the structs it stores internally: gvk::animated_node, gvk::bone_mesh_data, gvk::position_key, gvk::rotation_key, gvk::scaling_key. This task also is a pretty tricky one, because there will be changes required to (or an alternative to) model_t::prepare_animation_for_meshes_into_strided_contiguous_memory => see "Deserializing animation" below.
- gvk::animation_clip_data as returned by model_t::load_animation_clip

Serializing/deserializing image data

I think, it would be optimal, if the actual IMAGE DATA which is loaded from file (gvk::create_image_from_file) or from memory (gvk::create_1px_texture) would also be cached. That means serializing/deserializing the following data:

- std::array<uint8_t, 4> which is passed to gvk::create_1px_texture => This is easy and would not even have to be invasive into the gvk::create_1px_texture function.
- stbi_uc* as loaded from file by stbi_load in material_image_helpers.hpp#L101
- float* as loaded from file by stbi_loadf in material_image_helpers.hpp#L136
- void* meaning whatever is returned from gli::texture::data, like used in material_image_helpers.hpp#L80 and material_image_helpers.hpp#L201)
- Serializing/deserializing the image data directly would have the big advantage of being completely independent of file paths, so this would be the perfect solution. Cache files could even be shared across different PCs which would not be possible if the actual file paths would still be required.

However, in order to pull this off, gvk::create_image_from_file would have to be altered or and alternative "*_cached" version would have to be developed.

Hint: When working on this, please also regard the information from issue #59 and check with the other team members if someone might be working on that issue, because it would result in changes to all these relevant functions.

Deserializing animation

When deserializing gvk::animation, special care must be taken. gvk::animation internally stores the TARGET MEMORY ADDRESS of bone matrices. I.e. after a bone matrix has been calculated, it will be updated at that target memory addresses. These target memory addresses are, of course, not valid anymore after being deserialized from file. Therefore, model_t::prepare_animation_for_meshes_into_strided_contiguous_memory would have to be altered, or an alternative "*_cached" version might be required, or maybe simply a "set_bone_matrices_targets" function would have to be implemented that sets the new target memory addresses.

Actually, probably three such "set_bone_matrices_targets" functions would be required:

- set_bone_matrix_targets_into_strided_contiguous_memory
- set_bone_matrix_targets_into_tightly_packed_contiguous_memory
- set_bone_matrix_targets_with_offsets
- When implementing this, check back with @johannesugb since there is a (rather big) chance that gvk::animation has to be extended a bit.

Use a serialization library:

Best idea would probably be to use a suitable serialization library. First of all, research a bit which options there are and select a suitable one. Ensure that the serialization library satisfies the following requirements:

- The project is still "alive"
- The license complies with Gears-Vk's which is licensed under MIT, i.e. the license of the serialization library must comply with MIT. I believe that there were some discrepancies between GPL and MIT, s.t. MIT software may not include GPL code? But I could be wrong about that => to be checked!
- Good usability
- Good performance

Definition of done:

- The serialization library has been added as external dependency (also to the Visual Studio project/property files).
- The post build helper's option "Always Deploy Release DLLs" always deploys the release DLLs and it still works. If you need to tweak the library's build settings, please compare with stb_as_library -- that project should have a properly set-up config.
- Vertex data can be serialized/deserialized, i.e. the data returned by the methods declared in material_image_helpers.hpp#L439ff

- Material data from gvk::convert_for_gpu_usage declared in material_image_helpers.hpp#L396 can be serialized/deserialized
- Lightsource data gvk::lightsource_gpu_data can be serialized/deserialized
- Animation data gvk::animation can be serialized/deserialized and some way to re-target bone matrix target pointers is implemented
- The orca_loader example has been modified to support caching, i.e. serializing all those resources to file on first usage and deserializing from cached files if they exist. In order to achieve that some general changes to the example will be required: 1) use gvk::get_vertices_and_indices (and manually upload to the GPU) instead of gvk::create_vertex_and_index_buffers, 2) use gvk::get_2d_texture_coordinates_flipped (and manually upload to the GPU) instead of gvk::create_2d_texture_coordinates_flipped_buffer, 3) use gvk::get_normals (and manually upload to the GPU) instead of gvk::create_normals_buffer, and using all the cache-supporting versions of all the functions, like the cached alternative of gvk::convert_for_gpu_usage. Please see the comment below before changing the **orca_loader** example!
- The file LICENSE.md has been updated with the information of the added library.

Serialization/deserialization functionality is well documented and the Contribution Guidelines have been followed.

Problem description

- 3D engines load assets from disk, memory, or network
- Loading requires
  - opening files, or network socket,
  - allocating memory on CPU side
  - read file/stream into memory,
  - decode raw data (e.g. Jpeg to sRGB, compressed mesh to array),
- possibly convert images/meshes/materials into another format/data type/memory layout,
- transform data (flip image), postprocess (compute normals, tangents)
- Finally data is copied into Vulkan resources on the GPU (buffers/images)

Caching intends to increase performance of resource loading by

- Storing final data to a cache file on disk after the first time it was loaded
- Loading data from the cache file instead of the original source on subsequent access

Caching can reduce the number of file accesses, memory allocations, memory transfers, data decoding, conversion, transformations, postprocessing.

Caching invariants

For caching to work correctly, the following conditions must hold between cache store and cache load operations:

- Original data files must not change. (i.e. content of asset files)
- The conversion, transformation, and postprocessing of data must not change. (i.e. operations, parameters to operations on data in program)
- The types to construct must not change (e.g. switching between different texture samplers)

If any of the conditions does not hold, the affected part of the cache file must be invalidated.

Cache file layout

- The order of data must not change. E.g. when program loads assets in different order (based on random variable or user input)

In the simplest case, all assets are loaded in a fixed order every time the program is run. With more complex applications, this assumption may not hold.

By assigning a unique ID to every asset, it is possible to store a map with the cache file and load the right asset as needed. This also allows it to invalidate individual assets without recreating the whole cache file.

Ideally, the ID would include a hash of the raw data, but this would require to hash the file every time the data is needed, which defeats the purpose of caching. A better option would be to include the file name and modification time.

It is also possible to create a separate cache file for each scene or model.

For best performance, every asset should be loaded at most once, e.g. if multiple models reference the same texture file.

Enabling/Disabling caching

Switching between implementation with and without caching:

- For debugging (infer if a bug is related to caching)
- Rapid development: assets may chance often during development, recreating a cache file every time negates the increased performance by caching;
- deleting a cache file manually adds an extra step in the test cycle, hence it may be more convenient to temporarily disable it completely;
- custom data types may change during development and it would be extra work to adapt the serialization methods each time

Enabling caching should be possible without a separate code path:

- Avoid redundancy
- Increase test coverage
- Easier adoption of caching

Is caching actually needed?

One could argue that caching is unnecessary if assets are already provided in a format that is most efficient for loading into GPU buffers. I.e. store all textures and meshes in the format used by the GPU, with optional decompression on the GPU. In addition, all assets can be placed in a single file archive and accessed by ID instead of file names. This essentially replaces caching with asset preprocessing and has several advantages:

- There is only a single path to load assets, considerably simplifying the program;
- Asset loading is optimized from the start, not only after the first run;
- Serialization of all assets is done in the preprocessing step, not during program execution;
- The archive never needs to be written to, reducing the risk of file corruption and allowing for storage on read-only media;
- The archive takes less disk space than the original assets + cache file (if all assets are cached);
- Assets can be updated without the need to monitor files on disk - simply map new data to the asset IDs used in an application (referring to assets using strings is usually considered a bad practice in 3D engines)

The major downside of preprocessing is the extra step in the asset pipeline:

- When an asset is updated, or the operations on it are changed, the preprocessing must be repeated.
- The information on data operations and data usage are kept in different locations - outside and inside the application. This increases the risk of inconsistencies and requires additional work to maintain.
- If an application uses different operations depending on the platform or settings, all variants of each asset must be included in the archive, and the asset loader must be able to distinguish between them.
- It is not possible to load user-provided assets unless the preprocessing tool is included in the application or there is a loader for these less efficient file formats.

Hybrid model: define asset loading operations in application, load only from archive, trigger preprocessing when asset is not found in archive or when all assets should be created (e.g. before release):

- Keep definitions of operations and usage in one place;
- Allows to load dynamically defined assets (e.g. user-provided, loaded from network)

- Preprocessing can be omitted from application for release version if not needed

Industry Alternatives to caching

XBox Velocity Architecture

- (see e.g. https://news.xbox.com/en-us/2020/07/14/a-closer-look-at-xbox-velocity-architecture/)
- Microsoft GameStack video (April 2021): https://www.youtube.com/watch?v=zolAIEH0n1c
- Only available as part of Microsoft DirectX 12 Ultimate, already in use on XBox Series X?
- optimized for NVME SSDs
- Three main aspects:
    - Hardware accelerated decompression: standard LZ compression and proprietary texture compression, DirectCompute based decompression on GPU
    - DirectStorage API: new low-level, high-performance, parallel, highly programmable IO from SSD to GPU (only available as developer preview for selected game studios as of summer 2021, to be released for PCs with Windows 11?)
    - Sampler Feedback Streaming: only load parts of mipmaps that are needed in scene (overview, specification and sample code are available here: https://devblogs.microsoft.com/directx/coming-to-directx-12-sampler-feedback-some-useful-once-hidden-data-unlocked/)

Nvidia RTX IO

- https://www.nvidia.com/en-us/geforce/news/rtx-io-gpu-accelerated-storage-technology/
- GPU-based lossless game asset decompression
- Plugs into DirectStorage API, enables reading data directly from SSD or NIC into GPU RAM
- Developer preview available for selected game studios
- Interoperability with Vulkan?

Caching in gears-vk framework

Overview

- Documented in https://github.com/cg-tuwien/Gears-Vk/blob/master/docs/serializer.md
- Basic usage: initialize archive, replace function calls with `*_cached` variants
- Only works when loading order and assets do not change between different runs
- Downsides:

- saving to/loading from archive is implemented in each function instead of per type: new functions that create a type must re-implement caching as well; no separation of concerns
- With caching, some function parameters are empty, e.g. `aVerticesAndIndices` for `create_vertex_and_index_buffers_cached()`
- User must have different code paths for loading with/without caching
- `*_cached functions` (2 per actual function) create namespace clutter
- Objects without a suitable constructor cannot be cached (~~e.g. GLI texture only loads from file~~)
- Caching code and all its dependencies is included even if not used

Basic usage

In general, when saving or using no cache:

- Data is loaded from an asset
- An instance of a type is constructed with certain parameters (parameters can be function arguments, but also be taken from input, e.g. `image_data`, or orca scene)
- The data is loaded into the instance
- The data and parameters are saved to the cache file

When loading:

- Parameters (only those not available as function arguments) and data are loaded from cache
- An instance of the return type is constructed with the parameters
- The data is loaded into the instance

Basic implementation pattern for value types (e.g. structs, vectors) in pseudocode:

- Function

```
create_type_x(params) {
      x = do_something(params);
      return x;
}
```

becomes

```
create_type_x_cached(serializer, params) {
      if (serializer.mode == serialize or serializer == none)
            x = do_something(params);
            serializer.save(x);
      } else {
            serializer.load(x);
      }
      return x;
}
```

Functions and parameters by data source

```
std::vector<std::tuple<avk::resource_reference<const
gvk::model_t> std::vector<mesh_index_t>>>&
aModelsAndSelectedMeshes
```

- `get_normals_cached(gvk::serializer& aSerializer, ...)`
- `create_normals_buffer_cached(gvk::serializer& aSerializer, ...)`
- etc.

Observations

- All `get_*_cached()` functions return containers of `glm::vec` data, retrieved from the respective models
- `aModelsAndSelectedMeshes` can be expressed with a Composite design pattern and has the same interface as `model_t`
- Eliminate the `get_*()` functions by making them methods of a `ModelsAndSelectedMeshes` Composite
- Add a `model_t` subtype (or decorator; decorator applies to individual instances) that caches all `get_*()` functions; alternatively, use the Strategy pattern to implement caching, i.e. `model_t` forwards loading/saving data to a strategy object.
- Enable caching by using the `model_t` subtype instead of the original type

Remarks

- (does `model_t` have to be a resource? Collection of plain data arrays)
- (`model_t` should not create buffers from per-vertex data - reduces coupling)
- (`load_from_file()` should not be a method of `model_t`? - single responsibility principle)

Creating buffers from per-vertex data:

- Without caching:
  `create_normals_buffer(...)` calls
  `get_normals(...)`, passes data to
  `create_buffer(...)`
- With caching, avoid extra copy of data:
  `create_normals_buffer_cached(...)`
  if serializing, loads data as above, saves it to cache
  If deserializing, calls `create_buffer_cached(...)`, given serializer

`create_buffer_cached(...)` loads data from cache into host visible buffer, creates buffer, and transfers it to GPU

If `create_*_buffer()` functions are methods of the `model_t` class, they can be overridden to implement caching directly in the `_cached` subclass.

Alternatively, do not pass the data itself to `create_*_buffer()`, but a buffer data object. This object knows the object/method that loads the data and has a `.data()` method just as a `std::vector` that returns a pointer to the data. Preferably these buffer data objects would only exist within the `model_t` class.

A cached buffer type can also use the buffer data object to implement caching: either load the data from the buffer data object or from the serializer, depending on the serializer mode.

std::vector<gvk::material_config> aMaterialConfigs

- `convert_for_gpu_usage_cached(gvk::serializer& aSerializer, ...)`

Image_data

- `create_image_from_image_data_cached(...)`
- `create_cubemap_from_image_data_cached(...)`
- `create_image_from_file_cached(...)`
- `create_cubemap_from_file_cached(...)`

The `image_data` class uses the PIMPL idiom to separate abstraction and implementor of `image_data_interface`. Users of the framework create instances of the abstraction, which contains a pointer to an implementation. This is to support dynamic selection of an implementation at runtime, i.e. switch between image loading libraries depending on file type. Currently the data stored in `image_data` is serialized/deserialized where it is used (e.g. in `create_image_from_image_data_cached()`).
Ideally the image_data type should be serialized as a whole instead to make functions processing image_data instances agnostic to caching. Generally this requires writing serializer functions for each implementor, however it might be possible to instead define a new type of implementor that loads all data from the cache.
A trivial implementation of this implementor, `image_data_cached`, would create an additional overhead reading cached data to memory that is then copied to Vulkan buffers. That is, to deserialize an instance of `image_data`, one would also have to construct and deserialize an implementor (basically a GLI texture or stb array).
To make the loading from cache more efficient, an approach similar to the `model_t` classes described above can be applied: define `image_data_cached as a` type that handles caching and buffer creation, or use buffer data objects that load data and fill buffers as needed.

Appendix A: model_t caching draft

Early draft demonstrating composite pattern and decorator to implement caching for model_t and create_normals_buffer(). Methods, parameters and specifiers (e.g. const) not relevant to the draft are omitted.

```
// base class of Composite hierarchy
class model_t_component
{
    // same interface as class model_t
    // abstract class, no or default implementation

public:
    // example interface, no implementation
    // Note: returning normals for multiple/all indices could probably be specified with another composite or selection type, but
this is not essential for this draft
    // for simplicity, assume we always use a vector of indices
    virtual std::vector<glm::vec3> normals_for_mesh(std::vector<mesh_index_t>& aMeshIndices) = 0; // return normals for one mesh

    // one way to create buffers for model
    virtual avk::buffer create_normals_buffer(std::vector<mesh_index_t>& aMeshIndices /*...*/)
    {
           // default implementation:
           // data = normals_for_mesh(aMeshIndices, ...);
           // return create_buffer(data, ...);
    }
};

class model_t_scene : public model_t_component
{
    // the class model_t, renamed
    // leaf, i.e. has no child components, directly returns results for create_*_buffer()

public:
    // constructor only stores arguments, does not load anything yet
    model_t_scene(/*...*/)
    {
           // store path, etc.
    };

    // implement interface
    // Note that virtual functions are only resolved when using pointers or references to objects, e.g. the following piece of
code always calls this method directly, without using the virtual function table:
    // model_t_scene m(/*...*/);
    // auto result = m.normals_for_mesh();
    virtual std::vector<glm::vec3> normals_for_mesh(std::vector<mesh_index_t>& aMeshIndices)
    {
           // load scene if not loaded yet
           // convert and return normal data
    };

    // use default implementation for
    // virtual avk::buffer create_normals_buffer(std::vector<mesh_index_t>& aMeshIndices /*...*/)
};

class model_t_composite : public model_t_component
{
    // basically a type for aModelsAndSelectedMeshes, i.e. it stores
    // std::vector<std::tuple<avk::resource_reference<const gvk::model_t_component>, std::vector<mesh_index_t>>>

public:
    model_t_composite(/*...*/)
    {
           // store aModelsAndSelectedMeshes
    };

    virtual std::vector<glm::vec3> normals_for_mesh(std::vector<mesh_index_t>& aMeshIndices)
    {
            // no need to check if loaded here, children will load themselves on demand
            // aggregate results of method over specified children
    };

    // the composite interface might include methods for handling components (add, remove, ...) if needed
};

// caching decorator (aka wrapper), individually applied to each _instance_ of a model_t_component
// basic usage:
// ser = Serializer("cachefile.bin");
// model_t_component m("myscene.orca");
// model_t_caching_decorator md(m, ser);
// md.normals_for_mesh(...) // md applies caching to all its methods
class model_t_caching_decorator : public model_t_component
{
    // implements model_t interface, caches all results from interface methods

public:
    model_t_caching_decorator(model_t_component& m, Serializer s)
    {
            // store component, serializer
    };

    virtual std::vector<glm::vec3> normals_for_mesh(std::vector<mesh_index_t>& aMeshIndices)
    {
            // no need to check if loaded here, component will load itself on demand
            // load and return result from cache if possible
            // otherwise get result from component, save in cache, and return
    };

    virtual avk::buffer create_normals_buffer(std::vector<mesh_index_t>& aMeshIndices /*...*/)
    {
            // load and create buffer from cache if possible
            // otherwise create buffer from component, save in cache, and return
    }
};

// the model_t_components can also be specialized to include caching, either reimplementing methods or using the decorator
// advantage: simpler construction (same as uncached type except serializer),
// downside: must specialize each type (leaf, composite) separately

// caching subclass based on decorator, used instead of non-caching type, uniformly applies caching to all of its instances
// basic usage:
// ser = Serializer("cachefile.bin");
// model_t_scene_cached mc("myscene.orca", ser);
// mc.normals_for_mesh(...) // mc applies caching to all its methods
class model_t_scene_cached : public model_t_caching_decorator
{
    model_t_scene m;
public:
    // same constructor as model_t_scene class, except serializer argument
    model_t_scene_cached(/*...*/, Serializer s) : m(/*...*/), model_t_caching_decorator(m, s)
    {
            // nothing to be done here
    };

    // The implementation of the component interface is reused from model_t_caching_decorator, no need to reimplement anything
here
};
```

Appendix B: C++ serialization library cereal overview

- Header-only, C++ 11 library
- Reversibly turns arbitrary data types into different encodings, such as binary, XML, JSON
- Supports most types from the standard library out of the box
- Supports smart pointers (raw pointers and references are not supported)
- Extensible for other types of archives and data types
- Easy to use
- Permissive license

Basic usage:

- Implement serialize method in every custom class (or separate load/save methods)
- Create instance of archive to write to/read from (typically a file on disk, but can be any std::iostream)
- Write to/read from archive
- For writing ensure archive destructor is called when finished to ensure all data is written

Additional basic usage options:

- Serialization methods can also be implemented outside of class (e.g. if source code is not available)
- Also supports class versioning, private serialization methods, classes without default constructor
- Naming values: objects can have a name attached to it (see below for details)

Considerations for application:

- It is the user's responsibility to ensure compatibility between saved and loaded archives. It is recommended to use the same version of cereal for loading and saving data.
- Class versioning can be used to support different implementations of classes
- Polymorphic types must be registered before they can be serialized
- To preserve endianness between different machines, use the portable binary archive

Serialization functions:

- Either serialize, save/load, or save_minimal/load_minimal
- Defined inside or outside of classes
- Class methods can be private if cereal::access is declared a friend.
- Optional versioning

- Serialization functions are templates that can be specialized for different types of archives
- The function can be renamed if required
- only one type of serialization function per data type/archive pair; use disambiguation if inheritance introduces two types
- Separate save/load used e.g. to dynamically allocate memory for loading

- Save function must be const or take const reference
- Minimal functions are mainly used to simplify output to human readable archives (XML, JSON)
- The available function is automatically determined by cereal
- External functions should be in the namespace of the types they serialize or in the cereal namespace
- For serializing smart pointers to types without a default constructor, use a special overload (see pointers section in docs)

Naming values:

- When writing to archives, a user can provide a name-value pair (NVP) for each object. The name can be the same as the variable name passed to the archiver, or a user-provided custom string. If no name is provided, cereal will automatically generate an enumerated name
- When loading, names are optional but can also be used out of order
- Only useful for human readable formats such as XML or JSON, binary data ignores NVPs
- Could be used for binary data by specializing a binary archive?

Pointers:

- Supports smart pointers and pointers to polymorphic types,
- raw pointers and references are not supported, these must be handled by the user on a case by case basis
- Support requires default constructor or specialization of cereal::LoadAndConstruct, or static load_and_construct method
- Cereal ensures data pointed to by shared_ptr is serialized only once
- Pointer serialization can be deferred for handling complex data types (e.g. graphs)

Inheritance:

- Use cereal::base_class to serialize a base class in a derived class (takes care of abstract base classes, different serialization functions)
- For virtual inheritance, use cereal::virtual_base_class

Archive Specialization, Thread Safety:

- See documentation for details

PIMPL (pointer-to-implementation) idiom

- For this construct it is necessary to define the serialization functions only after the class of the implementation has been declared. As a consequence, each serialization function must be explicitly instantiated for each archive type (See https://uscilab.github.io/cereal/pimpl.html for details).