DISMAL

Drawing/Input/Sound Meta-Abstraction Layer

Technical Specification and User's Guide

Contents

Part I.

Specification

1. Goals

DISMAL (an acronym for Drawing/Input/Sound Meta-Abstraction Layer) is an attempt to make a small, highly portable game engine framework for 2D games programmed in C.

DISMAL is termed a *meta-abstraction layer* as it often relies on the work of other abstraction layers, such as SDL, to interface with the target hardware.

The layer will provide a more high-level outlook than most of its backends, hiding the complexity of creating a screen, handling input, managing resources and providing timing features.

1.1. Specific Targets

1.1.1. Generic

- The layer must be written in compliant C89, with the possible exception of code expected to be compiled by one compiler (for example, target driver code).
- The layer must run on Windows 95 and above.
- For the first milestone, the layer must be sufficient to run a SameGame clone on target code for SDL, protected mode DOS and AmigaOS 3.1.
- The entire code base must be well-documented through use of both regular and Doxygen special comments.

1.1.2. Modules

- The layer code will be organised into modules. For the first milestone, these will be selected at compile-time by use of conditional compilation.
- A *base* module will be produced for each target family. Each other module will depend on a particular base, to allow large libraries to share init/deinit code and prevent unworkable module combinations.
- For the first milestone, three base modules will be created: SDL (Simple Direct-Media Layer 1.2, multi-platform), Amiga68k (AmigaOS 3.x and above on 68020 and above processors) and DOS (32-bit extended DOS on 80386 and above).
- The following other module types will eventually be produced:

  - GFX (low-level graphics interface, including image resource handler)
  - SFX (sound and music interface)
  - Input (keyboard and mouse interface)
  - State (finite state machine engine)
  - GUI (a more high-level user interface layer)
  - Time (timing and FPS, if available)
  - Config (argument vector, config file and Windows registry reader)

- In addition to the modules, a small amount of DISMAL's platform-independent functionality will be kept outside the module system and compiled into all programs. This includes DISMAL's main init/deinit code.

1.1.3. GFX

- The layer must be able to report to its parent code if the GFX target is a low-resolution target. Low-resolution is defined as 320x200 8-bit.

- The layer must be able to adapt graphics intended for 320x200 8-bit to higher resolutions if the parent code cannot. This includes double-scaling graphics and co-ordinates and letterboxing.

- All targets of the layer must support the loading of 256-colour PCX files, with an optional transparent palette index (either given explicitly or stated as a RGB triple to match, for example magenta 255, 0, 255), as a fallback when specific target image files are not present.

- The layer must always provide a logical palette of 256 colours, which is attached to the physical palette in low-resolution modes.

- The layer will work in terms of 8bpp colour components, quantising these when necessary (for example to 6bpp VGA).

- The PCX handler should be able to crudely remap colours to the existing logical palette by comparing average colour values.

1.1.4. Sound

- All targets should be capable of playing back ProTracker Module (.mod) format files as music.

1.1.5. Input

- The input layer will make use of callbacks to provide input events to other parts of the code. These callbacks will be provided along with input filters, so that only relevant information triggers them (for example, mouse button code can request that only mouse events trigger a callback).

- ASCII key events and non-ASCII ("special") key events will be segregated. The layer will pass all ASCII key events unmodified, but special keys will be passed instead as constants referring to their meaning (eg Escape key, arrow keys, Enter key). These will trigger separate callbacks.

1.1.6. State

- DISMAL will provide a generic, well-commented main loop, pulling in all enabled modules using #ifdef, which can be used directly in code or copied and pasted into the parent program's logic and modified.

- DISMAL will provide the framework for a finite state machine model of application programming, using function pointers to achieve polymorphism without the use of C++.

Part II.

Technical

Part III.

User's Guide

2. Preparation

DISMAL is, at the time of writing, maintained and distributed as a directory of source code with the intention of being included in a project and compiled as part of the project.

Therefore, in order to use DISMAL, place it in the directory tree of the DISMAL-using project, and chain the DISMAL makefile into the project makefile, or otherwise incorporate the necessary DISMAL sources into the project compiling process.

3. Initialisation and deinitialisation

3.0.7. Initialising DISMAL

The easiest way to use DISMAL is with code similar to the following:

```
if (dm_init() == DM_SUCCESS) {
  ...code goes here...
  dm_cleanup();
} else {
  fprintf(stderr, "Could not initiate DISMAL!\n");
}
```

Figure 3.1.: DISMAL initialisation code.

What is, or is not, initialised depends on how DISMAL was compiled. By default, all compiled-in modules will be initialised.

In order to gain finer control over the loading of modules, one can use the function `dm_set_module_list` to change the module list. An example is given below.

```
dm_set_module_list(DM_MOD_GFX | DM_MOD_SFX); /* Init GFX and sound only. */

if (dm_init() == DM_SUCCESS) {
  /* Code goes here */
  dm_cleanup();
} else {
  fprintf(stderr, "Could not initiate DISMAL!\n");
}
```

Figure 3.2.: DISMAL initialisation code with a custom configuration.

Note: If `init_modules` includes a module but not its prerequisite modules (for example, UI but not GFX), those modules will automatically be loaded.

Note: DISMAL returns `DM_SUCCESS` if there were no fatal errors reached while loading. A `DM_SUCCESS` during init does not necessarily mean everything has loaded - see the next secion.

3.0.8. Getting information about DISMAL

One of the first things you may want to do after initialising DISMAL is to check a few things about the engine state. DISMAL exports its current status through several functions, documented here.

DISMAL regards errors loading the SFX module as non-fatal - that is, `dm_init` will report `DM_SUCCESS` even if the SFX loader has failed. This is because sound is generally an optional extra in games (which is DISMAL's intended purpose), whereas most other modules are critical to game functions. However, all other module failures will cause DISMAL to fail to load.

Therefore, you may wish to check to see if the sound module has loaded and, if not, why not. This can be done by checking the output of the function `dm_get_module_list`. This will return an integer with the lags for each initialised module set.

```
int modlist;

modlist = dm_get_module_list();

if (modlist & DM_MOD_SFX) {
  /* Sound code goes here */
} else {
  /* No sound */
}
```

Figure 3.3.: Checking to see if the sound module loaded.

4. Using the graphics subsystem

4.1. High-res and low-res

Due to DISMAL targetting widely disparate

4.2. Loading images

DISMAL provides two methods for loading an image. The first