XLA Specification v1.0.0

Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [@!RFC2119] [@!RFC8174] when, and only when, they appear in all capitals, as shown here.

- `Archive`: refers to all the files, that are packed into `XLA` format
- `Compiler`: refers to a program, which reads (parses) `XLA` format files
- `Level code`: refers to raw level code, as per PewPew API documentation
- `Lua`: refers to a modified version of Lua programming language, used for creating PewPew levels, according to `PewPew API`
- `Manifest`: refers to `manifest.xml` file in the root of the `Archive`
- `PewPew API` refers to official PewPew API
- `XLA`: refers to a file format, which this spec describes.

Find definitions for terms `Preset`, `Transform` and `Patch` in the `Presets / Transforms / Patches` section intro.

Structure

`XLA` is a complex file format. Any assets are stored alongside data, for which XML format is used. A tarball is created from those, to pack everything into one file.

The general structure of files inside the tarball is as follows:

```
.
  <level>.xla/
      plugins/
          <plugin1>.lua
          <plugin2>.lua
          <plugin3>.lua
      mesh.xml
      entity.xml
      level.xml
      manifest.xml
      .xilia
```

Please refer to the rest of sections in this document for additional info on each one of the elements of this file structure.

Note on XML format

It is RECOMMENDED to specify XML version in `.xml` files, like so:

```
<?xml version="1.0"?>
```

This might be used by the `Compiler` to correctly parse the files.

Also note that the spec makes heavy use of XML namespaces, they must be a valid IRI.

Metadata file (`.xilia`)

In the root of the level, `.xilia` file MUST be present. That file contains the version of this spec, which is used for that particular `Archive`. The format is as following:

`v1.0.0-ud`

Include 6 or more characters. Start the version in `.xilia` file with the character `v`, then the version itself (semantic versioning v2 is used).

This will give you file contents like `v1.0.0`. If you want to add any flags, include a hyphen after the version number, and list the flags together, one letter for each.

The given above example, `v1.0.0-ud`, indicates using the version `1.0.0` for the project, and also includes both `u` and `d` flags.

Version flags

Following are the flags, that you can use to modify spec version:

- `u`: Unicode characters may be used in parts of the `Archive`
- `d`: A ”loose” variation of a spec. In this version, many values can be left undefined (it is up to `compiler` to set defaults).
- `g`: In this version, value types (in transforms/patches) may be ommited (it is up to `compiler` to figure out the type).

It is RECOMMENDED to list the flags in the same order, as they appear in this list.

If the flag can be applied to the `Archive`, you MUST include it.

Level manifest (`manifest.xml`)

Following is the example of how `manifest.xml` file might look like:

```
<?xml version="1.0"?>

<manifest xmlns="xilia://manifest">

  <xilia/>

  <level>
```

```
    <name>Eskiv</name>
    <descriptions>
      <description>Red definitely good.</description>
    </descriptions>
    <information/>

    <rank-thresholds>
      <rank-thresholds-1p>
        <bronze>2500</bronze>
        <silver>3500</silver>
        <gold>4500</gold>
      </rank-thresholds-1p>
      <rank-thresholds-2p/>
    </rank-thresholds>

  </level>

</manifest>
```

Refer to `Structure` section, `Note on XML format` subsection for additional info on XML standard used.

You MUST include `manifest` tag of `xilia://manifest` namespace in the `Manifest`. There SHOULD NOT be any other tags in the `Manifest`, except for `manifest` and `xml`.

Note that all the described in the spec elements MUST inherit the namespace of `xilia://manifest` from the `manifest` tag.

`manifest` tag MUST include `xilia` and `level` tags, and SHOULD NOT include any other tags. In current spec version the `xilia` tag is unused, so you MAY leave it empty.

Basic level info (`level` tag)

The `level` tag contains some basic information about the level, and closely resembles `manifest.json` file, as per official PewPew API docs

Please refer to the given above documentation for the meaning of each field. Following is the mapping between XML in `level` tag, and fields in `manifest.json` of the `Level code`:

- `name`: level name (string), maps to `name` in `manifest.json`.
- `descriptions`: MUST contain one or more `description` tags. SHOULD NOT contain any other tags. Each `description` tag is a string. Maps to `descriptions` array in `manifest.json`
- `information`: level information (string), maps to `information` in `manifest.json`.

- `entry-point`: path to the main entry point file. This is ultimately decided by `Compiler`, and including this field is OPTIONAL. If the `entry-point` field is present, it MAY be used by `Compiler`.
- `rank-thresholds`: MUST contain one and only one `rank-thresholds-1p` tag, as well as one and only one `rank-thresholds-2p` tag. In current spec both tags MAY be empty. `rank-thresholds-2p` SHOULD be ignored by `Compiler`, while empty `rank-thresholds-1p` tag maps to value `false` of `has_score_leaderboard` tag in `manifest.json`. If `rank-thresholds-1p` is not empty, it MUST contain `bronze`, `silver` and `gold` tags (all are integers), which map to `rank_thresholds_1p` in `manifest.json`. If `rank-thresholds-1p` is not empty, but all three values for `bronze`, `silver` and `gold` are 0, the level will not be casual, but also will not give any stars (like `Just Pong`).

Unless stated otherwise, all the mentioned tags MUST NOT be empty in standard spec version. If the spec version used includes `d` flag (see `Metadata file` section, `Version flags` subsection), fields with no required children MAY be empty.

Note that `PewPew API` imposes lenght limits on some of the fields. Those are not imposed by this spec, instead it is left up to the `Compiler`.

Presets / Transforms / Patches

One of the key aspects of how XLA describes levels, are elements. Elements can represent a mesh, or a piece of level logic, such as "What happens, when this enemy collides with a wall" or "How often, where and how this enemy spawns".

Each element MUST consist of a single preset, zero or more transforms and zero or more patches:

1. `Preset` is a base building block for the mesh/logic. For meshes it is a general mash shape, such as simple "circle", "square", "cube", "sphere", or more complex "baf-like", "rolling-sphere-like", etc.. For logic it describes the general behaviour with almost no set values (constants), like "Bounce of the walls on collision" or "Enemy spawns regularly in several set places on the map" (note how the "set places on the map" and "regularly" is not defined by a preset). Each preset exposes different variables/constants to change the visuals/behaviour. Some presets are built-in (refer to `xla-builtin.md` - W.I.P.), however other can be created with `Lua` and included into `Archive`.
2. `Patch` is a single change in a constant (not variable!), defined by a preset. Many patches can be applied to a single preset, while single patch can only be applied to a single preset. For every constant in a preset, there MUST be only one patch, modifying that constant. Each patch is declared as a key-value pair, where key is the constant to be changed (exposed by preset), and the value is a new value for that constant.

3. `Transform` is a more complex version of a patch. They can modify both variables and constants of presets. An example of a transform would be modifying a mesh preset to create multiple copies of it, or making an enemy rainbow (with animation). Some transforms are built-in (refer to `xla-builtin.md` - W.I.P.), however other can be created with `Lua` and included into `Archive`.

Defining elements in XML

Every element, apart from patches/transforms and a preset, MUST also include an id (string or integer value, which is up to `Archive` creator to decide); that id MUST me unique across all the elements in a given `Archive` (even across meshes/logic/etc).

Presets do not include any additional info, except for the preset name itself (for built-in presets) or name of the preset asset (for custom presets).

The following is the example of defining a complex element (mesh for Eskiv level border), using presets, transforms, and patches:

```
<element id="eskiv-border" xmlns="xilia://element">

  <preset>mesh.trivial.rectangle</preset>

  <transforms>

    <transform>
      <name>mesh.effect.clone-and-fade</name>
      <settings>
        <setting>
          <key>MULTIPLIER</key>
          <value type="number">13</value>
        </setting>
        <setting>
          <key>COLOR_START</key>
          <value type="color">15597823</value>
        </setting>
        <setting>
          <key>COLOR_END</key>
          <value type="color">2097407</value>
        </setting>
      </settings>
    </transform>

    <transform>
      <name>mesh.clone.clone-around</name>
      <settings>
```

```
        <setting>
          <key>MULTIPLIER</key>
          <value type="string">8</value>
        </setting>
        <setting>
          <key>ROTATE</key>
          <value type="bool">0</value>
        </setting>
        <setting>
          <key>DISTANCE</key>
          <value type="fx">50</value>
        </setting>
      </settings>
    </transform>

  </transforms>

  <patches>
    <patch>
      <key>WIDTH</key>
      <value type="fx">1000</value>
    </patch>
    <patch>
      <key>HEIGHT</key>
      <value type="fx">700</value>
    </patch>
    <patch>
      <key>COLOR</key>
      <value type="color">16711935</value>
    </patch>
  </patches>

</element>
```

Refer to `Structure` section, `Note on XML format` subsection for additional info on XML standard used.

Spec for creating elements Start creating an element with an `element` tag, which MUST be that of a `xilia://element` namespace, and all the child elements MUST inherit that namespace. The `element` tag MUST also include an `id` propery with a UNIQUE name, among all the elements (even different types) in the `Archive`.

Inside an `element` tag, there MUST be one and only one `preset` tag, zero or more `transforms` tags, zero or more `patches` tags. There SHOULD NOT be any other tags present.

Note that the order, in which changes to preset are applied, MUST be:

1. Transforms (`Compiler` MAY apply them in the order, how they appear in XML)
2. Patches (`Compiler` MAY apply them in the order, how they appear in XML)

Tags `preset`, `transforms` and `patches` MAY be in any order.

Describing preset The following is the example of a `preset` tag:

```
<preset>mesh.enemy.baf</preset>
```

A `preset` tag MUST NOT be empty, and it MUST contain a valid preset name.

Built-in preset name consists of 3 parts, which are separated by a period (`.`). Following are those parts:

- Preset type, can be one of: `mesh`, `logic`, `spawn`.
- Preset category
- Preset name

You can find the full list of presets from categories `mesh`, `logic` and `spawn` in `xla-builtin.md` doc.

Custom preset name consists of 2 parts, which are separated by a period (`.`). Following are those parts:

- Preset type, MUST be `custom`.
- Preset name

Read more about creating and using custom presets in `Custom presets / Transforms` subsection of this section. *After that, take a look at* ***`lua-spec.md`***

Describing transformations Having `transforms` tag is OPTIONAL, there MAY be zero or more of them. Each one of those MAY have zero or more `transform` tags. Together, all the `transform` tags in all the `transforms` tags define a set of transforms to apply to the preset.

Following is the example of `transform` tag:

```
<transform>
  <name>mesh.effect.bold</name>
  <settings>
    <setting>
      <key>MULTIPLIER</key>
      <value type="number">5</value>
    </setting>
  </settings>
</transform>
```

Each `transform` tag MUST contain `name` tag, with the transform name specified. Rules for how the transform names are formatted and what exactly they are, are the same, as for presets, so refer to `Describing preset` subsection.

`transform` tag can contain zero or more `settings` tags. Each one of those can contain zero or more `setting` tags. Together, all the `setting` tags in all the `settings` tags define a set of settings to apply to the transform.

Following is the example of `setting` tag:

```
<setting>
  <key>MULTIPLIER</key>
  <value type="number">2</value>
</setting>
```

Each setting MUST contain `key` and `value` tags. Key is a constant, exposed by a transform, and `value` is the new value for that constant.

Describing patches Having `patches` tag is OPTIONAL, there MAY be zero or more of them. Each one of those MAY have zero or more `patch` tags. Together, all the `patch` tags in all the `patches` tags define a set of patches to apply to the preset.

Following is the example of `patch` tag:

```
<patch>
  <key>COLOR</key>
  <value type="color">538976511</value>
</patch>
```

Each setting is formatted similarly to transforms, it MUST contain `key` and `value` tags. Key is a constant, exposed by a preset, and `value` is the new value for that constant.

Value types Any specified for transform or patch value MUST have a `type` attribute, attached to it (unless the `g` flag is set for the `Archive` spec version). In that case, in XLA, type MAY be ommited, and it is up to `Compiler` to figure out a type. Note that `Compiler` might not support XLA spec versions with `g` flag, so if all the types are know, it is recommended to list them and not include the `g` flag.

The available types are following (mainly mapping to `Lua` types):

- `string`: Any text, MAY be Unicode if `u` flag is present in `Archive` spec version. Otherwise MUST be ASCII. Maps to a `string` in `Lua`
- `number`: Integer or float, maps to `number` in `Lua`. No limits on size are enforced by this spec, that is left up to the `Compiler`.
- `fx`: Fixed point value, maps to `FixedPoint` type (class) in `Lua`.
- `color`: Refers to a color. It is expressed as a hex RGBA value in `Lua`, however XLA format uses RGBA integer (decimal) value.

If a type is specified for an element, it MUST be one of the stated above types.