

第1章：はじめに – Blenderアドオン開発の魅力と可能性

Blenderは、オープンソースの3D制作ソフトとして世界中のクリエイターに愛されています。モデリング、アニメーション、レンダリング、VFXなど、多岐にわたる機能を備えている一方で、標準機能だけではカバーしきれない作業もあります。そこで活躍するのが「Blenderアドオン」です。

アドオンを開発すれば、**作業を自動化**したり、**新しい機能を追加**したり、**Blenderを自分好みにカスタマイズ**することができます。

特に、スクリプトやプラグイン開発の経験がなくても、Pythonを使えば比較的簡単にアドオンを作成できます。

このシリーズでは、**Blenderアドオンの開発から公開、収益化までの全プロセス**を徹底解説します。

初心者の方でもゼロから学び、最終的には**自作アドオンを世界に向けて公開**することができるようになります。

1.1 Blenderアドオンとは？

Blenderアドオンとは、Pythonで作成されたスクリプト（プラグイン）で、Blenderの機能を拡張するためのものです。

例えば、以下のような用途で活用されています。

- **操作の自動化**：面倒な手順をワンクリックで実行
- **新しいツールの追加**：オリジナルのモデリングツールやリギング機能の実装
- **カスタムUIの作成**：BlenderのUIを自分好みにカスタマイズ
- **ワークフローの改善**：制作のスピードアップや効率化

Blenderには標準でも数多くのアドオンが搭載されており、[編集 > プリファレンス > アドオン](#) から有効化できます。

ただし、特定の用途に特化したアドオンは自作するのが最も理想的です。

1.2 Blenderアドオン開発の魅力

Blenderアドオンを開発することで、単なる3Dアーティストやアニメーターとしてだけでなく、**クリエイターゼンプログラマー**としてのスキルを身につけることができます。

以下のようなメリットがあります。

① 自分の作業を効率化できる

手作業で何十回も繰り返している作業を、**1クリックで自動化**できたらどうでしょうか？

例えば、以下のような操作はPythonスクリプトを使えば簡単に自動化できます。

- すべてのオブジェクトを自動でリネーム
- 一定間隔でオブジェクトを配置
- シーン内の不要なデータを削除して軽量化

アドオン開発を学ぶことで、**Blenderを自分専用のツールに進化**させることができます。

② 他のBlenderユーザーと共有できる

自作のアドオンをGitHubやBlender Marketで公開すれば、世界中のBlenderユーザーと共有できます。特に、Blenderはオープンソースの文化が根付いており、無料アドオンの配布も盛んです。優れたアドオンを開発すれば、コミュニティ内で注目を集め、多くのユーザーに使ってもらえるチャンスがあります。

③ 収益化が可能

Blenderアドオンは無料公開するだけでなく、**有料販売**することもできます。

例えば、以下のような販売プラットフォームを活用できます。

- **Blender Market** (公式マーケット)
- **Gumroad** (個人販売向け)
- **BOOTH** (日本向け)
- **Patreon / Fanbox** (継続的支援)

実際に、Blender Marketでは**数千ドル以上の売上を出しているアドオン**も珍しくありません。

特に、プロ向けの高度なツールを開発すれば、**安定した収益源**になる可能性があります。

1.3 Pythonを使う理由

Blenderアドオンの開発には、プログラミング言語「Python」を使用します。

Pythonは初心者でも学びやすく、簡潔な文法が特徴です。

```
print("Hello, Blender!")
```

この1行のコードを実行するだけで、BlenderのPythonコンソールに「Hello, Blender!」と表示されます。

また、PythonはBlender内部のデータを直接操作できるAPI (**bpy**) を備えており、以下のような処理が可能です。

- シーン内のオブジェクトを取得・変更する
- カスタムツールを追加する
- Blenderのインターフェースを改変する

Pythonの基本がわかれば、すぐにアドオン開発に取り組めるので、**これからプログラミングを学ぶ人にも最適**です。

1.4 Blenderアドオン開発の流れ

このシリーズでは、以下の流れでアドオン開発を進めます。

1. 環境構築 (Pythonの基礎、Blenderのスクリプトエディタの使い方)
2. 基本的なアドオンの作成 (フォルダ構成、**bl_info**の設定、**register()**の仕組み)
3. オペレーターとパネルの作成 (新しいボタンや機能の追加)
4. データの管理 (オブジェクトの操作、カスタムプロパティ)
5. UIの拡張 (カスタムメニュー、リスト、スライダーの作成)
6. テストとデバッグ (エラーハンドリング、開発効率を上げる方法)

7. アドオンの配布・公開 (GitHub、Blender Market、BOOTHなど)
8. 収益化の戦略 (無料配布 vs 有料販売、マーケティングのポイント)

最終的には、自分のアドオンを世界中に公開し、Blenderユーザーに役立つツールを作ることを目指します。

1.5 次回予告

次回は「**Blenderアドオンの基本構造とPythonの基礎**」を解説します。

実際にアドオンのフォルダ構成を作りながら、最初のスクリプトを実行してみましょう。

Blenderをさらに便利にする「自分だけのツール」を作る旅へ、さっそく出発しましょう！

第2章：Blenderアドオンの基本構造とPythonの基礎

前回の記事では、Blenderアドオンの魅力や可能性について紹介しました。

今回から実際に**アドオンを作るための基礎知識**を学び、最初のアドオンを作成してみましょう。

この章では、以下のポイントを解説します。

- Blenderアドオンのフォルダ構成
 - アドオンの必須ファイル `__init__.py` の書き方
 - `bl_info` とは？
 - `register()` / `unregister()` の仕組み
 - Pythonの基礎とBlender APIの簡単な使い方
-

2.1 Blenderアドオンのフォルダ構成

Blenderのアドオンは、**Pythonのスクリプト (.py)** として作成されます。

基本的には以下の2つの方法でアドオンを作ることができます。

1. 単一のPythonファイル

- 小規模なアドオンや簡単なスクリプト向け
- 1つの `.py` ファイルとして作成

2. フォルダ構成のパッケージ型

- 機能が多い場合や複数のファイルを扱う場合に適用
- `__init__.py` を含むフォルダ形式で作成

シンプルなアドオンなら、Pythonファイル1つでも動作しますが、**今後の拡張を考えるとフォルダ構成の方が管理しやすいです**。

Blenderの標準アドオンもフォルダ型を採用しているため、本格的に開発するならフォルダ構成を選ぶのがベストです。

標準的なフォルダ構成（推奨）

my_addon/	← アドオンのフォルダ（任意の名前）
__ __init__.py	← アドオンのエントリーポイント（必須）
__ operators.py	← 操作（オペレーター）を定義
__ ui.py	← UI（パネルやメニュー）を定義
__ utils.py	← 汎用関数やヘルパー関数を定義
└__ icons/	← カスタムアイコンを格納（必要なら）

アドオンのフォルダは**Pythonのモジュール（パッケージ）として機能**するため、`__init__.py` を必ず作成します。

これにより、フォルダごとBlenderに認識させることができます。

2.2 `__init__.py` の基本

`__init__.py` はアドオンの**エントリーポイント**であり、Blenderがアドオンを読み込む際に最初に実行されます。

最低限、以下の要素を含める必要があります。

- **アドオンの情報 (`bl_info`)**
- **アドオンの登録 (`register()`)**
- **アドオンの解除 (`unregister()`)**

実際のコード例を見てみましょう。

```
bl_info = {
    "name": "My First Addon",
    "author": "あなたの名前",
    "version": (1, 0, 0),
    "blender": (3, 0, 0),
    "location": "View3D > Tool Shelf",
    "description": "Blenderにボタンを追加するシンプルなアドオン",
    "category": "Object",
}

import bpy

def register():
    print("My First Addon が有効になりました")

def unregister():
    print("My First Addon が無効になりました")
```

bl_info の各項目

`bl_info` は、Blenderのアドオン一覧に表示される**メタデータ（基本情報）**を定義します。

- `name` → アドオンの名前（Blenderのアドオンリストで表示される）
- `author` → 開発者の名前

- **version** → アドオンのバージョン ((**major**, **minor**, **patch**) の形式)
- **blender** → 対応するBlenderのバージョン
- **location** → アドオンが追加される場所 (例: 「3Dビューのツールシェルフ」)
- **description** → アドオンの説明 (簡潔に)
- **category** → カテゴリー (Object、Mesh、Renderなど)

この **bl_info** を正しく記述しないと、アドオンが正しく認識されないので注意しましょう。

2.3 register() / unregister() の仕組み

Blenderのアドオンは、ユーザーが「有効化」すると **register()** が実行され、「無効化」すると **unregister()** が実行されます。

register() の役割

- 必要なオペレーター (機能) やパネル (UI) を登録
- イベントハンドラーを設定
- カスタムプロパティを定義

unregister() の役割

- すべての登録を解除
- 追加したカスタムプロパティを削除

以下のように、**register()** で追加したものは、**unregister()** で必ず解除するのが基本です。

```
import bpy

class MyOperator(bpy.types.Operator):
    """サンプルオペレーター"""
    bl_idname = "object.my_operator"
    bl_label = "My Operator"

    def execute(self, context):
        print("ボタンが押されました！")
        return {'FINISHED'}

    def register():
        bpy.utils.register_class(MyOperator)

    def unregister():
        bpy.utils.unregister_class(MyOperator)
```

ポイント

- **bpy.utils.register_class(MyOperator)** でオペレーターを登録
- **bpy.utils.unregister_class(MyOperator)** で解除 (しないとエラーが出る)

2.4 Pythonの基礎とBlender APIの簡単な使い方

Blenderアドオン開発ではPythonの基本的な文法を知っておく必要があります。

① 変数とデータ型

Pythonの基本的なデータ型は以下のようになります。

```
num = 10          # 整数
text = "Blender" # 文字列
flag = True       # 真偽値
```

② 関数の定義

関数を作成すると、処理をまとめて簡単に再利用できます。

```
def greet():
    print("こんにちは！Blenderアドオン開発へようこそ")

greet() # 関数を呼び出す
```

③ Blender APIの基本

Blenderでは `bpy` モジュールを使って内部データにアクセスできます。

現在のシーンのオブジェクト一覧を取得

```
import bpy

for obj in bpy.data.objects:
    print(obj.name)
```

新しいキューブを追加

```
bpy.ops.mesh.primitive_cube_add()
```

オブジェクトの位置を変更

```
bpy.context.object.location.x += 2
```

このように `bpy` を使うことで、Blender内部のデータやオブジェクトを直接操作できます。

2.5 まとめ

この章では、Blenderアドオンの基本的な構造とPythonの基礎を学びました。

- アドオンは `__init__.py` を含むフォルダで作成する
- `bl_info` で アドオンの情報 を定義する
- `register()` で 機能を登録し、 `unregister()` で解除 する
- `bpy` モジュールを使えば **Blenderのデータを操作** できる

2.6 次回予告

次回は、**「オペレーター（Operator）を作成し、ボタンを追加する方法」**を解説します。
実際にBlenderに新しいボタンを作り、クリックすると何かが起こるアドオンを作ってみましょう！

第3章：オペレーター（Operator）を作成し、ボタンを追加する

前回の記事では、Blenderアドオンの基本構造やPythonの基礎、`register()` と `unregister()` の仕組みを学びました。

今回からは実際に **Blenderに新しいボタン（オペレーター）を追加** して、アドオンとしての形を作っています。

この章では、以下の内容を解説します。

- オペレーター（Operator）とは？
- オペレーターの基本構造
- ボタンを3Dビューに追加
- オペレーターの実用例
- ショートカットキーの登録

3.1 オペレーター（Operator）とは？

Blenderにおける「オペレーター」とは、ユーザーが **ボタンやショートカットキーを押したときに実行される処理**のことです。

例えば、以下のような操作はすべて「オペレーター」として実装されています。

- オブジェクトの追加 (`bpy.ops.mesh.primitive_cube_add()`)
- オブジェクトの移動・回転 (`bpy.ops.transform.translate()`)
- レンダリングの実行 (`bpy.ops.render.render()`)
- エクスポート (`bpy.ops.export_scene.obj()`)

これらのオペレーターは `bpy.ops` の中に定義されており、Pythonスクリプトから直接呼び出すことができます。

```
import bpy
```

```
# キューブを追加するオペレーターを実行  
bpy.ops.mesh.primitive_cube_add()
```

Blenderの標準オペレーターだけでなく、**自分でオリジナルのオペレーターを作成することも可能**です。

3.2 オペレーターの基本構造

オペレーターを作成するには、`bpy.types.Operator` を継承したクラスを定義し、`execute()` メソッドの中に処理を書きます。

シンプルなオペレーターの例

```
import bpy  
  
class SimpleOperator(bpy.types.Operator):  
    """オペレーターの説明（ツールチップに表示される）"""  
    bl_idname = "object.simple_operator" # 一意の識別子（カテゴリー.名前）  
    bl_label = "シンプルなボタン" # UI上に表示されるラベル  
  
    def execute(self, context):  
        # ここにオペレーターの処理を書く  
        print("ボタンが押されました！")  
        return {'FINISHED'} # 成功時の戻り値
```

オペレーターを作成する際のポイント

- `bl_idname` → 一意の識別子（"カテゴリー.オペレーター名" の形式）
- `bl_label` → UIに表示される名前
- `execute(self, context)` → ボタンが押されたときに実行される処理

register() に追加

オペレーターを作成したら、`register()` で登録し、`unregister()` で解除する必要があります。

```
def register():  
    bpy.utils.register_class(SimpleOperator)  
  
def unregister():  
    bpy.utils.unregister_class(SimpleOperator)
```

この状態でBlenderのPythonコンソールから以下のようにオペレーターを実行できます。

```
bpy.ops.object.simple_operator()
```

3.3 3Dビューにボタンを追加

オペレーターを作成しただけでは、ユーザーがGUIから実行できません。
そこで、3Dビューのツールシェルフにボタンを追加してみましょう。

パネルを作成し、ボタンを追加

```
import bpy

class SimpleOperator(bpy.types.Operator):
    """オペレーターの説明"""
    bl_idname = "object.simple_operator"
    bl_label = "シンプルなボタン"

    def execute(self, context):
        print("ボタンが押されました！")
        return {'FINISHED'}


class SimplePanel(bpy.types.Panel):
    """3Dビューにボタンを追加するパネル"""
    bl_label = "カスタムツール"
    bl_idname = "VIEW3D_PT_custom_tools"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "ツール"

    def draw(self, context):
        layout = self.layout
        layout.operator("object.simple_operator")

def register():
    bpy.utils.register_class(SimpleOperator)
    bpy.utils.register_class(SimplePanel)

def unregister():
    bpy.utils.unregister_class(SimpleOperator)
    bpy.utils.unregister_class(SimplePanel)

if __name__ == "__main__":
    register()
```

ポイント

- `bl_space_type = 'VIEW_3D'` → 3DビューのUIに表示
- `bl_region_type = 'UI'` → サイドバーに配置
- `bl_category = "ツール"` → タブの名前
- `layout.operator("object.simple_operator")` → ボタンを追加

実行方法

1. スクリプトをBlenderのテキストエディターで実行

2. 「ツール」タブを開くと「シンプルなボタン」が表示される
 3. ボタンをクリックすると、コンソールに **ボタンが押されました！** と表示される
-

3.4 実用的なオペレーター

次に、**実用的なオペレーター** を作成してみましょう。

例えば、「選択したオブジェクトをX方向に2m移動する」オペレーターを作ります。

選択オブジェクトを移動するオペレーター

```
class MoveObjectOperator(bpy.types.Operator):  
    """選択オブジェクトを移動"""  
    bl_idname = "object.move_x"  
    bl_label = "X軸に2m移動"  
  
    def execute(self, context):  
        obj = context.object # アクティブなオブジェクトを取得  
        if obj:  
            obj.location.x += 2  
            self.report({'INFO'}, "オブジェクトを移動しました")  
        else:  
            self.report({'WARNING'}, "オブジェクトを選択してください")  
        return {'FINISHED'}
```

このオペレーターを登録し、パネルにボタンを追加すると、クリックするだけで選択オブジェクトがX方向に2m移動するようになります。

3.5 ショートカットキーの登録

Blenderでは、オペレーターに**ショートカットキーを割り当てる** こともできます。

ショートカットキーを登録

```
addon_keymaps = []  
  
def register():  
    bpy.utils.register_class(MoveObjectOperator)  
  
    wm = bpy.context.window_manager  
    km = wm.keyconfigs.addon.keymaps.new(name="3D View", space_type="VIEW_3D")  
    kmi = km.keymap_items.new("object.move_x", type='M', value='PRESS', ctrl=True)  
    addon_keymaps.append((km, kmi))  
  
def unregister():  
    bpy.utils.unregister_class(MoveObjectOperator)  
  
    for km, kmi in addon_keymaps:
```

```
km.keymap_items.remove(kmi)
addon_keymaps.clear()
```

このコードを追加すると、**Ctrl + M** でオブジェクトをX方向に2m移動できるようになります。

3.6まとめ

この章では、オペレーターの作成方法と、ボタン・ショートカットキーの登録方法を学びました。

- `bpy.types.Operator` を継承してオペレーターを作成
- `execute()` の中で処理を実装
- `layout.operator()` でボタンをUIに追加
- ショートカットキーを登録し、素早くオペレーターを実行可能にする

3.7 次回予告

次回は「**BlenderのUIをカスタマイズし、パネルやメニューを作成する方法**」を解説します！

さらに使いやすいアドオンを作るため、UI設計の基本を学びましょう！

第4章：BlenderのUIをカスタマイズし、パネルやメニューを作成する

前回は **オペレーター（Operator）** を作成し、3Dビューにボタンを追加しました。

今回は **BlenderのUIをカスタマイズ** し、より使いやすいインターフェースを作成する方法を解説します。

この章では、以下の内容を学びます。

- **BlenderのUIの基本構造**
- **パネル（Panel）の作成**
- **カスタムメニューの追加**
- **リスト表示（UIList）の活用**
- **UIのデザインとユーザビリティの向上**

4.1 BlenderのUIの基本構造

BlenderのUIは、**スペース（space）** と**リージョン（region）** に分かれています。

UI要素	説明
スペース (<code>bl_space_type</code>)	どのエリアに表示するか（例: <code>VIEW_3D</code> , <code>PROPERTIES</code> ）
リージョン (<code>bl_region_type</code>)	どの部分に表示するか（例: <code>UI</code> , <code>TOOLS</code> , <code>HEADER</code> ）
カテゴリー (<code>bl_category</code>)	サイドバーのタブ名（3Dビュー用）
パネル (<code>bpy.types.Panel</code>)	UIを構成する基本単位

4.2 パネル（Panel）の作成

4.2.1 3Dビューにカスタムパネルを追加

まず、3Dビューのサイドバー（ツールシェルフ）に新しいタブを作成し、オペレーターを追加してみましょう。

```
import bpy

class CustomPanel(bpy.types.Panel):
    """3Dビューにカスタムパネルを追加"""
    bl_label = "カスタムツール" # パネルタイトル
    bl_idname = "VIEW3D_PT_custom_tools"
    bl_space_type = 'VIEW_3D' # 3Dビューに表示
    bl_region_type = 'UI' # サイドバー (Nパネル)
    bl_category = "ツール" # タブ名

    def draw(self, context):
        layout = self.layout
        layout.label(text="オペレーターを実行") # テキスト表示
        layout.operator("object.move_x") # オペレーターのボタンを追加

class MoveObjectOperator(bpy.types.Operator):
    """オブジェクトをX方向に移動"""
    bl_idname = "object.move_x"
    bl_label = "X軸に2m移動"

    def execute(self, context):
        obj = context.object
        if obj:
            obj.location.x += 2
            self.report({'INFO'}, "オブジェクトを移動しました")
        else:
            self.report({'WARNING'}, "オブジェクトを選択してください")
        return {'FINISHED'}

def register():
    bpy.utils.register_class(CustomPanel)
    bpy.utils.register_class(MoveObjectOperator)

def unregister():
    bpy.utils.unregister_class(CustomPanel)
    bpy.utils.unregister_class(MoveObjectOperator)

if __name__ == "__main__":
    register()
```

4.2.2 実行方法

1. Blenderのテキストエディタでスクリプトを実行
2. Nキーを押してサイドバーを開く
3. 「ツール」タブに「カスタムツール」パネルが追加される
4. 「X軸に2m移動」ボタンを押すと、選択オブジェクトが移動する

4.3 カスタムメニューの追加

パネルだけでなく、**メニューを拡張する** こともできます。

4.3.1 メニューを追加

以下のコードを追加すると、オブジェクトモードの右クリックメニューに「X軸に2m移動」の項目を追加できます。

```
class CustomMenu(bpy.types.Menu):
    """カスタムメニュー"""
    bl_label = "カスタム操作"
    bl_idname = "OBJECT_MT_custom_menu"

    def draw(self, context):
        layout = self.layout
        layout.operator("object.move_x")

    def menu_func(self, context):
        self.layout.menu("OBJECT_MT_custom_menu")

    def register():
        bpy.utils.register_class(CustomMenu)
        bpy.utils.register_class(MoveObjectOperator)
        bpy.types.VIEW3D_MT_object.append(menu_func)

    def unregister():
        bpy.utils.unregister_class(CustomMenu)
        bpy.utils.unregister_class(MoveObjectOperator)
        bpy.types.VIEW3D_MT_object.remove(menu_func)
```

4.3.2 実行方法

1. スクリプトを実行
2. 3Dビューで **オブジェクトを右クリック**
3. 「カスタム操作」 → 「X軸に2m移動」の項目が追加される

4.4 リスト表示 (UIList) の活用

オブジェクトのリストやカスタムデータを表示する場合、**bpy.types.UIList** を使うことができます。

4.4.1 カスタムオブジェクトリストを作成

```
class OBJECT_UL_custom_list(bpy.types.UIList):
    """オブジェクトのリスト"""
    def draw_item(self, context, layout, data, item, icon, active_data,
                 active_propname, index):
```

```

        layout.label(text=item.name, icon='OBJECT_DATAMODE')

class CustomListPanel(bpy.types.Panel):
    """リストを表示するパネル"""
    bl_label = "オブジェクトリスト"
    bl_idname = "VIEW3D_PT_custom_list"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "ツール"

    def draw(self, context):
        layout = self.layout
        layout.template_list("OBJECT_UL_custom_list", "", context.scene,
"objects", context.scene, "active_object")

def register():
    bpy.utils.register_class(OBJECT_UL_custom_list)
    bpy.utils.register_class(CustomListPanel)

def unregister():
    bpy.utils.unregister_class(OBJECT_UL_custom_list)
    bpy.utils.unregister_class(CustomListPanel)

```

4.4.2 実行方法

1. スクリプトを実行
2. **N**キーでサイドバーを開く
3. 「ツール」タブにオブジェクトリストが表示される
4. リストからオブジェクトを選択すると、アクティブオブジェクトが変わる

4.5 UIのデザインとユーザビリティの向上

BlenderのUIをカスタマイズする際、以下のポイントを意識すると **ユーザビリティが向上**します。

4.5.1 パネルの整理

- 関連する機能を1つのパネルにまとめる
- 複数のカテゴリにまたがらないようにする
- セクションをボックス (`layout.box()`) で分ける

```

box = layout.box()
box.label(text="基本操作")
box.operator("object.move_x")

```

4.5.2 UIのレイアウト

- 縦に並べる → `layout.prop()`
- 横に並べる → `row = layout.row(); row.prop(...)`

- カラムを作成 → `col = layout.column(); col.prop(...)`

```
row = layout.row()
row.operator("object.move_x")
row.operator("object.delete")
```

4.5.3 ツールチップの活用

- `bl_description` を追加すると、ツールチップが表示される

```
class MoveObjectOperator(bpy.types.Operator):
    """選択オブジェクトをX方向に2m移動"""
    bl_idname = "object.move_x"
    bl_label = "X軸に2m移動"
    bl_description = "選択オブジェクトをX方向に2m移動します"
```

4.6 まとめ

この章では、BlenderのUIをカスタマイズし、パネルやメニューを追加する方法を学びました。

- `bpy.types.Panel` を使って サイドバーにパネルを追加
- `bpy.types.Menu` で 右クリックメニューにカスタム項目を追加
- `bpy.types.UIList` を使って オブジェクトのリストを表示
- UIの整理とデザインのコツ

4.7 次回予告

次回は 「Blenderのデータ構造とカスタムプロパティの管理」 を解説します。

アドオンで独自の設定を保存し、データを管理する方法を学びましょう！

第5章：Blenderのデータ構造とカスタムプロパティの管理

前回は **BlenderのUIをカスタマイズし、パネルやメニューを追加する方法** を学びました。

今回は、Blenderの内部データ構造を理解し、**カスタムプロパティ（独自のデータ）を管理する方法** を解説します。

この章では、以下の内容を学びます。

- **Blenderのデータ構造（シーン・オブジェクト・プロパティ）**
- **カスタムプロパティの追加**
- **カスタムプロパティをパネルで編集**
- **カスタムプロパティの保存と読み込み**
- **プロパティグループの活用**

5.1 Blenderのデータ構造とは？

Blenderのデータは、**データブロック (Data Blocks)** という構造で管理されています。

データブロックは、Blenderの**.blend** ファイルに保存され、**相互にリンク可能** です。

例えば、以下のようなデータ構造を持っています。

```
Scene (シーン)
└─ Collection (コレクション)
    └─ Object (オブジェクト)
        └─ Mesh (メッシュデータ)
        └─ Material (マテリアル)
        └─ Modifier (モディファイア)
        └─ Custom Properties (カスタムプロパティ)
        └─ Constraints (コンストRAINT)
    └─ Light (ライト)
└─ Camera (カメラ)
└─ World (ワールド設定 )
└─ Render Settings (レンダー設定 )
└─ Custom Properties (シーンのカスタムプロパティ)
```

データアクセスの例

```
import bpy

# シーンにある全オブジェクトの名前を表示
for obj in bpy.data.objects:
    print(obj.name)

# アクティブオブジェクトのメッシュデータを取得
mesh = bpy.context.object.data if bpy.context.object.type == 'MESH' else None

# シーンのレンダー解像度を変更
bpy.context.scene.render.resolution_x = 1920
bpy.context.scene.render.resolution_y = 1080
```

Blenderのデータ構造を理解すると、**シーンやオブジェクトの情報をカスタマイズ** しやすくなります。

5.2 カスタムプロパティとは？

カスタムプロパティ (Custom Properties) は、オブジェクトやシーンに独自のデータを追加する機能です。

例えば、「**オブジェクトの分類**」「**数値設定**」「**カスタムタグ**」などを保存できます。

カスタムプロパティを使うことで、**アドオンで独自のデータを管理** したり、Blenderの既存機能を拡張することが可能になります。

5.3 オブジェクトにカスタムプロパティを追加

オブジェクトに「カスタムID」というプロパティを追加してみましょう。

5.3.1 カスタムプロパティの追加

```
import bpy

bpy.context.object["custom_id"] = 123 # カスタムプロパティを追加
print(bpy.context.object["custom_id"]) # 取得
```

オブジェクトにプロパティが追加される場所

1. **N**キーでサイドバーを開く
2. 「オブジェクト」タブを選択
3. 「カスタムプロパティ」セクションに `custom_id` が表示される

5.4 パネルでカスタムプロパティを編集

カスタムプロパティをパネルで編集できるようにしてみましょう。

```
import bpy

class CustomPropertyPanel(bpy.types.Panel):
    """カスタムプロパティ編集パネル"""
    bl_label = "カスタムプロパティ"
    bl_idname = "VIEW3D_PT_custom_props"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "ツール"

    def draw(self, context):
        layout = self.layout
        obj = context.object

        if obj:
            layout.prop(obj, '[ "custom_id"]', text="カスタムID")
        else:
            layout.label(text="オブジェクトを選択してください")

    def register():
        bpy.utils.register_class(CustomPropertyPanel)

    def unregister():
        bpy.utils.unregister_class(CustomPropertyPanel)

if __name__ == "__main__":
    register()
```

実行方法

1. スクリプトを実行
 2. **N**キーでサイドバーを開く
 3. 「ツール」タブに「カスタムプロパティ」が表示される
 4. カスタムIDを変更すると、オブジェクトにデータが保存される
-

5.5 プロパティグループを活用

Blenderの `bpy.props` を使うと、より便利にプロパティを管理できます。

5.5.1 プロパティグループの作成

```
import bpy

class CustomProperties(bpy.types.PropertyGroup):
    custom_id: bpy.props.IntProperty(name="カスタムID", default=100)
    custom_text: bpy.props.StringProperty(name="カスタムテキスト", default="デフォルト")
    custom_toggle: bpy.props.BoolProperty(name="オン/オフ", default=False)

def register():
    bpy.utils.register_class(CustomProperties)
    bpy.types.Object.my_props = bpy.props.PointerProperty(type=CustomProperties)

def unregister():
    bpy.utils.unregister_class(CustomProperties)
    del bpy.types.Object.my_props

if __name__ == "__main__":
    register()
```

このスクリプトを実行すると、オブジェクトに `my_props` というカスタムプロパティグループが追加されます。

5.5.2 プロパティグループをパネルで編集

```
class CustomPropertyPanel(bpy.types.Panel):
    """カスタムプロパティパネル"""
    bl_label = "カスタムプロパティ"
    bl_idname = "VIEW3D_PT_custom_props"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "ツール"

    def draw(self, context):
        layout = self.layout
        obj = context.object
```

```
if obj and hasattr(obj, "my_props"):
    layout.prop(obj.my_props, "custom_id")
    layout.prop(obj.my_props, "custom_text")
    layout.prop(obj.my_props, "custom_toggle")
else:
    layout.label(text="オブジェクトを選択してください")

def register():
    bpy.utils.register_class(CustomPropertyPanel)

def unregister():
    bpy.utils.unregister_class(CustomPropertyPanel)

if __name__ == "__main__":
    register()
```

実行方法

1. スクリプトを実行
 2. **N** キーでサイドバーを開く
 3. 「ツール」タブ → 「カスタムプロパティ」に新しい項目が追加される
-

5.6 まとめ

この章では、Blenderのデータ構造とカスタムプロパティの管理方法を学びました。

- Blenderのデータは **データブロック** で管理されている
 - カスタムプロパティを追加すると、**オブジェクトに独自のデータを保存できる**
 - **bpy.props** を使うと、より便利にプロパティを管理できる
 - UIパネルを作成して、カスタムプロパティを **GUIで編集できるようにする**
-

5.7 次回予告

次回は「アドオンの設定を保存・読み込みする方法」を解説します！
ユーザーの設定を永続化し、アドオンを便利にする方法を学びましょう！

第6章：アドオンの設定を保存・読み込みする方法

前回は **Blenderのデータ構造とカスタムプロパティの管理**について学びました。
今回は、**アドオンの設定を保存・読み込みする方法**を解説します。

Blenderでは、アドオンの設定を **Blenderのプリファレンス（ユーザー設定）**に保存できます。
これにより、**Blenderを再起動しても設定が維持される**ようになります。

この章では、以下の内容を学びます。

- **アドオンの設定を保存する方法**

- アドオンの設定をUIパネルで編集
 - アドオンの設定を読み込む
 - 設定をデフォルト値にリセット
 - 設定をファイルにエクスポート・インポート
-

6.1 アドオンの設定を保存する方法

アドオンの設定を保存するには、`bpy.types.AddonPreferences` を使うのが最も簡単な方法です。

Blenderの「編集」>「プリファレンス」>「アドオン」に設定を追加できます。

6.1.1 基本的なアドオンの設定

以下のコードを実行すると、アドオンの設定をプリファレンスに保存できます。

```
import bpy

class MyAddonPreferences(bpy.types.AddonPreferences):
    """アドオンの設定を管理"""
    bl_idname = __name__ # アドオンID（通常は __name__ を使う）

    my_string: bpy.props.StringProperty(name="文字列", default="デフォルト値")
    my_int: bpy.props.IntProperty(name="数値", default=10)
    my_bool: bpy.props.BoolProperty(name="スイッチ", default=True)

    def draw(self, context):
        layout = self.layout
        layout.label(text="アドオンの設定")
        layout.prop(self, "my_string")
        layout.prop(self, "my_int")
        layout.prop(self, "my_bool")

    def register():
        bpy.utils.register_class(MyAddonPreferences)

    def unregister():
        bpy.utils.unregister_class(MyAddonPreferences)

    if __name__ == "__main__":
        register()
```

6.1.2 実行方法

1. スクリプトを実行
2. 「編集」>「プリファレンス」>「アドオン」を開く
3. 検索バーに `MyAddonPreferences` を入力
4. 「アドオンの設定」という項目が追加されている

この方法を使うと、アドオンの設定を**Blenderのプリファレンス**に保存できます。

6.2 アドオンの設定をUIパネルで編集

プリファレンスに保存した設定を、**3Dビューのパネル**から編集できるようにしてみましょう。

```
import bpy

class MyAddonPreferences(bpy.types.AddonPreferences):
    """アドオンの設定を管理"""
    bl_idname = __name__

    my_string: bpy.props.StringProperty(name="文字列", default="デフォルト値")
    my_int: bpy.props.IntProperty(name="数値", default=10)
    my_bool: bpy.props.BoolProperty(name="スイッチ", default=True)

    def draw(self, context):
        layout = self.layout
        layout.label(text="アドオンの設定")
        layout.prop(self, "my_string")
        layout.prop(self, "my_int")
        layout.prop(self, "my_bool")

class MyAddonPanel(bpy.types.Panel):
    """3Dビューに設定パネルを追加"""
    bl_label = "アドオン設定"
    bl_idname = "VIEW3D_PT_myaddon_panel"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "ツール"

    def draw(self, context):
        layout = self.layout
        prefs = bpy.context.preferences.addons[__name__].preferences

        layout.label(text="アドオン設定")
        layout.prop(prefs, "my_string")
        layout.prop(prefs, "my_int")
        layout.prop(prefs, "my_bool")

def register():
    bpy.utils.register_class(MyAddonPreferences)
    bpy.utils.register_class(MyAddonPanel)

def unregister():
    bpy.utils.unregister_class(MyAddonPreferences)
    bpy.utils.unregister_class(MyAddonPanel)

if __name__ == "__main__":
    register()
```

6.2.1 実行方法

1. スクリプトを実行
2. **N** キーでサイドバーを開く
3. 「ツール」タブの「アドオン設定」パネルが表示される
4. 設定を変更すると、プリファレンスに反映される

6.3 設定を読み込む

アドオンが起動したときに、**プリファレンスの設定を読み込む** 方法を解説します。

6.3.1 設定の取得

プリファレンスに保存された値をスクリプト内で取得できます。

```
prefs = bpy.context.preferences.addons['__name__'].preferences
print(prefs.my_string)
print(prefs.my_int)
print(prefs.my_bool)
```

これを活用すれば、アドオンの挙動をユーザー設定に応じて変更できます。

6.4 設定をデフォルト値にリセット

設定をデフォルト値に戻すボタンを追加してみましょう。

6.4.1 設定をリセットするオペレーター

```
class ResetPreferencesOperator(bpy.types.Operator):
    """設定をリセットする"""
    bl_idname = "addon.reset_preferences"
    bl_label = "設定をリセット"

    def execute(self, context):
        prefs = bpy.context.preferences.addons['__name__'].preferences
        prefs.my_string = "デフォルト値"
        prefs.my_int = 10
        prefs.my_bool = True
        self.report({'INFO'}, "設定をリセットしました")
        return {'FINISHED'}

    def register():
        bpy.utils.register_class(ResetPreferencesOperator)

    def unregister():
        bpy.utils.unregister_class(ResetPreferencesOperator)
```

6.4.2 リセットボタンをパネルに追加

```
layout.operator("addon.reset_preferences", text="リセット")
```

6.5 設定をファイルにエクスポート・インポート

プリファレンスの設定を **JSONファイル** に保存し、後で読み込めるようにしてみましょう。

6.5.1 設定をJSONに保存

```
import json
import bpy

class SaveSettingsOperator(bpy.types.Operator):
    """設定をJSONファイルに保存"""
    bl_idname = "addon.save_settings"
    bl_label = "設定を保存"

    def execute(self, context):
        prefs = bpy.context.preferences.addons[__name__].preferences
        settings = {
            "my_string": prefs.my_string,
            "my_int": prefs.my_int,
            "my_bool": prefs.my_bool
        }
        with open("/tmp/addon_settings.json", "w") as f:
            json.dump(settings, f)
        self.report({'INFO'}, "設定を保存しました")
        return {'FINISHED'}
```

6.5.2 設定をJSONから読み込み

```
class LoadSettingsOperator(bpy.types.Operator):
    """設定をJSONファイルから読み込む"""
    bl_idname = "addon.load_settings"
    bl_label = "設定を読み込む"

    def execute(self, context):
        prefs = bpy.context.preferences.addons[__name__].preferences
        try:
            with open("/tmp/addon_settings.json", "r") as f:
                settings = json.load(f)
            prefs.my_string = settings["my_string"]
            prefs.my_int = settings["my_int"]
            prefs.my_bool = settings["my_bool"]
        except:
            self.report({'WARNING'}, "設定を読み込めませんでした")
```

```
    self.report({ 'INFO' }, "設定を読み込みました")
except FileNotFoundError:
    self.report({ 'ERROR' }, "設定ファイルが見つかりません")
return {'FINISHED'}
```

6.6 まとめ

この章では、アドオンの設定を保存・読み込みする方法を学びました。

- **Blenderのプリファレンスに設定を保存**
- **UIパネルから設定を編集**
- **設定を読み込んでアドオンの動作に反映**
- **デフォルト値にリセットする機能**
- **設定をJSONファイルに保存・読み込み**

6.7 次回予告

次回は「アドオンを配布・公開する方法」を解説します！

GitHubやBlender Marketにアドオンをアップロードし、世界中のユーザーに届ける方法を学びましょう！

第7章：アドオンを配布・公開する方法

前回はアドオンの設定を保存・読み込みする方法を学びました。

今回は、作成したアドオンを配布・公開する方法を解説します。

Blenderアドオンを公開することで、他のユーザーにも使ってもらえたり、**収益化**することも可能になります。

今回は、以下の内容を詳しく解説します。

- **アドオンの配布準備**
- **ZIPファイルにパッケージ化**
- **GitHubで公開**
- **Blender Marketで販売**
- **Gumroad / BOOTH で配布**
- **公式アドオンリポジトリへの登録**
- **アップデートとメンテナンスの重要性**

7.1 アドオンの配布準備

まず、アドオンを他のユーザーに提供する前に、**必要なファイルを整理**しておきましょう。

7.1.1 アドオンのフォルダ構成（推奨）

```
my_addon/
  └── __init__.py           ← アドオンフォルダ（英数字が望ましい）
                                ← メインスクリプト（必須）
```

operators.py	← 操作を定義（オプション）
ui.py	← UIを定義（オプション）
utils.py	← ヘルパー関数（オプション）
README.md	← 説明書（推奨）
LICENSE	← ライセンス情報（推奨）
example.blend	← サンプルファイル（オプション）
docs/	← ドキュメント（オプション）

重要なファイル

- `__init__.py` → アドオンのエントリーポイント（必須）
- `README.md` → 使い方を記載
- `LICENSE` → ライセンス情報（MIT, GPL など）

7.2 ZIPファイルにパッケージ化

Blenderのアドオンは ZIP形式で配布するのが一般的です。

7.2.1 ZIP圧縮手順

1. アドオンフォルダ（`my_addon/`）を選択
2. 右クリック > 圧縮 > ZIP形式
3. `my_addon.zip` を作成

このZIPファイルを「アドオンをインストール」から簡単に追加できます。

7.2.2 Blenderでインストールテスト

1. Blenderを開く
2. 「編集」 > 「プリファレンス」 > 「アドオン」
3. 「インストール」ボタンをクリック
4. `my_addon.zip` を選択
5. インストール後、アドオンを有効化して動作確認

7.3 GitHubで公開

オープンソースとして無料で公開する場合、GitHubを利用すると便利です。

7.3.1 GitHubにアップロード

1. GitHubにログインし、新しいリポジトリを作成
2. リポジトリ名を `my_addon` にする
3. `README.md` や `LICENSE` を追加
4. ローカルのアドオンフォルダをGit管理下に追加

```
git init
git add .
git commit -m "First release"
```

```
git branch -M main  
git remote add origin https://github.com/ユーザー名/my_addon.git  
git push -u origin main
```

7.3.2 GitHub Releases でZIPを公開

1. リポジトリの「Releases」タブに移動
2. 「New Release」ボタンをクリック
3. バージョンを指定（例: v1.0.0）
4. `my_addon.zip` をアップロード
5. 「Publish Release」ボタンをクリック

この方法で **ユーザーが簡単にZIPをダウンロード** できるようになります。

7.4 Blender Marketで販売

Blender Market は、有料アドオンを販売できるプラットフォームです。

7.4.1 Blender Marketでの販売手順

1. アカウントを作成
2. 「Sell Your Work」から販売者登録
3. 新しい商品を追加
4. ZIPファイルをアップロード
5. 価格を設定（\$5～\$50が一般的）
6. 説明文とスクリーンショットを追加
7. 販売開始！

Blender Marketの特徴

- 手数料30%
 - 多くのBlenderユーザーが利用
 - セールやプロモーションが可能
 - Blender開発の支援にもつながる
-

7.5 Gumroad / BOOTH で配布

Blender Market以外でも、**Gumroad** や **BOOTH** を利用して販売できます。

7.5.1 Gumroadでの販売

Gumroad は、デジタル商品を販売できるプラットフォームです。

1. **Gumroad**に登録
2. 「New Product」を作成
3. ZIPファイルをアップロード
4. 価格を設定
5. 販売ページを公開

メリット

- 手数料が低い（8-10%）
 - 最低価格0円にして投げ銭方式も可能
 - 購入者リストを管理できる
-

7.5.2 BOOTHでの販売

BOOTHは、日本向けの販売プラットフォームです。

1. Pixivアカウントでログイン
2. ショップを作成
3. ZIPファイルをアップロード
4. 価格を設定
5. 販売開始！

メリット

- 日本円で販売できる
 - コンビニ決済や銀行振込が可能
 - クリエイター向けのコミュニティ
-

7.6 Blender公式リポジトリへの登録

Blenderの公式アドオンとして登録するには、[Blender Add-ons公式リポジトリ](#)に申請する必要があります。

7.6.1 公式アドオンの登録手順

1. GitHubでコードを公開
2. Blenderの開発者向けガイドラインを満たす
3. Blenderの開発者サイトに登録
4. 「New Task」を作成し、提案を提出
5. コードレビューを受ける
6. 承認後、Blender公式に追加される

公式アドオンの基準

- BlenderのUIガイドラインに準拠
 - 適切なライセンス（GPL）を適用
 - 品質チェックをクリア
 - 他のアドオンと競合しない
-

7.7 アップデートとメンテナンスの重要性

アドオンを公開したら、定期的にアップデートすることが重要です。

7.7.1 バージョン管理

- 1.0.0 → 最初のリリース
- 1.1.0 → 機能追加
- 1.2.0 → バグ修正

バージョンを明記し、GitHubのリリースノートに変更点を記載 しましょう。

7.7.2 ユーザーフィードバックを活用

- GitHub Issues でバグ報告を受ける
- Blender Marketのレビューを参考に改善
- TwitterやDiscordでユーザーの意見を聞く

7.8 まとめ

この章では、アドオンの配布・公開方法を学びました。

- ZIPファイルにパッケージ化
- GitHubで無料公開
- Blender Marketで販売
- Gumroad / BOOTH で配布
- 公式アドオンリポジトリに登録
- アップデートとメンテナンスの重要性

7.9 次回予告

次回は「アドオンの収益化戦略」を解説します！

有料アドオンの価格設定やマーケティング手法を詳しく見ていきましょう！

第8章：アドオンの収益化戦略

前回はアドオンの配布・公開方法について学びました。

今回は、アドオンを収益化する戦略について詳しく解説します。

Blenderアドオンは、無料公開だけでなく、有料販売によって収益を得ることも可能です。

実際に、Blender MarketやGumroadで月に数千ドル以上の売上を出しているアドオン開発者も存在します。

この章では、以下の内容を学びます。

- 有料アドオン vs 無料アドオン
- 価格設定の考え方
- 販売プラットフォームの比較
- 効果的なマーケティング戦略
- ユーザーサポートとコミュニティの構築
- アップデートと長期的な収益化の重要性

8.1 有料アドオン vs 無料アドオン

まず、アドオンを **有料で販売するか、無料で公開するか** を考えましょう。

項目	無料アドオン	有料アドオン
ユーザー数	多い	限定的（価格による）
収益	なし（寄付やスポンサー）	直接収益化
サポートの負担	軽い（放置でもOK）	重い（サポート義務が発生）
継続的な開発	モチベーション維持が難しい	収益があれば開発継続しやすい
ブランド力	コミュニティで広まりやすい	プロダクトとして評価される

どちらを選ぶかは、**アドオンの規模やターゲット層** によります。

無料公開が向いている場合

- ・ **シンプルな機能** のアドオン（小規模ツール）
- ・ コミュニティに貢献したい場合
- ・ オープンソース開発 をしたい場合
- ・ 寄付（Patreon, Ko-fi） やスポンサーを得たい場合

有料販売が向いている場合

- ・ **高度な機能** を持つアドオン
- ・ 開発やサポートに時間をかける予定がある
- ・ 継続的なアップデートを提供したい
- ・ **収益化を目的にする**

8.2 値格設定の考え方

アドオンの価格設定は **ユーザーにとっての価値** に基づくべきです。

8.2.1 一般的な価格帯

アドオンの規模	価格の目安	例
小規模ツール	\$1 ~ \$10	シンプルなワークフロー改善アドオン
中規模アドオン	\$10 ~ \$30	モデリング補助、アニメーションツール
高度なアドオン	\$30 ~ \$100	プロ向けの高度なリグシステムやレンダリングツール
プレミアム	\$100以上	企業やスタジオ向けのソリューション

価格を決める際は、**市場の競合アドオンと比較** するのが重要です。

例えば、同じ機能の無料アドオンがあるなら価格を低めに設定する必要があるでしょう。

8.3 販売プラットフォームの比較

有料アドオンを販売する場合、どのプラットフォームを使うかも重要です。

8.3.1 Blender Market

- URL: <https://blendermarket.com/>
- 特徴:
 - Blender専用のマーケット
 - 多くのユーザーが利用
 - 販売手数料 30%
 - アップデート・サポートが求められる
 - Blender開発基金の支援につながる
- おすすめのアドオン: プロ向けツール、リギング・モデリング支援

8.3.2 Gumroad

- URL: <https://gumroad.com/>
- 特徴:
 - 手数料が低い (8-10%)
 - 価格を「0ドル以上（投げ銭）」に設定可能
 - ユーザーリストを管理できる
 - 自分でプロモーションが必要
- おすすめのアドオン: 個人開発のツール、手軽な販売

8.3.3 BOOTH（日本向け）

- URL: <https://booth.pm/>
- 特徴:
 - 日本円で販売可能
 - Pixivアカウントがあれば簡単に販売できる
 - コンビニ決済・銀行振込対応
 - 3D系のクリエイターが多い
- おすすめのアドオン: 日本人ユーザー向けのツール

8.4 効果的なマーケティング戦略

アドオンを販売するだけでは、すぐには売れません。

適切なマーケティング戦略 を使うことで、多くのユーザーに知ってもらうことが重要です。

8.4.1 プロモーションの方法

1. デモ動画を作成 (YouTube, Twitter)
2. Blender Artists や Discordで宣伝
3. Blender Nation に記事を投稿
4. アフィリエイト (Blender Market) を活用
5. 期間限定セールを実施

8.4.2 セールの活用

Blender Marketでは、**ブラックフライデー** や **ホリデーシーズン** に **20-50%オフのセール** を実施すると、売上が大きく伸びることが多いです。

例:

- 通常価格 \$20 → セール価格 \$15 (25%オフ)
 - 販売数が2倍になれば、売上も増加
-

8.5 ユーザーサポートとコミュニティの構築

有料アドオンを販売するなら、**サポート体制を整えることが重要** です。

8.5.1 サポートの種類

- GitHub Issues** → バグ報告を受け付ける
- Discordサーバーを開設** → ユーザーと交流する
- FAQ (よくある質問) を用意** → 質問対応の負担を減らす

サポートをしっかりとすると、高評価レビューがつきやすくなり、売上アップにつながるので、できる範囲で対応しましょう。

8.6 アップデートと長期的な収益化

一度アドオンを販売したら終わりではなく、**定期的にアップデートを提供することが重要** です。

8.6.1 アップデートのメリット

- バグ修正でユーザー満足度アップ
- 新機能追加で再購入促進
- Blenderのバージョンアップに対応

例えば、バージョンを「**1.0 → 1.1**」にするだけでも、ユーザーに**「開発が続いている」と安心感を与えられる**ので、積極的にアップデートしましょう。

8.7 まとめ

この章では、アドオンの収益化戦略について学びました。

- 無料 vs 有料アドオンの選択**
 - 価格設定の考え方**
 - 販売プラットフォームの比較**
 - マーケティング戦略とプロモーション**
 - サポート体制とコミュニティの構築**
 - アップデートと長期的な収益化**
-

8.8 次回予告

次回は「**Blenderアドオン開発の高度なテクニック**」を解説します！

より複雑なアドオンの開発に挑戦してみましょう！

第9章 : Blenderアドオン開発の高度なテクニック

前回は**アドオンの収益化戦略**について解説しました。

今回は、より高度なアドオン開発に挑戦するための**テクニック**を紹介します。

この章では、以下の内容を学びます。

- カスタム描画（OpenGL / GPU）
 - Blenderの依存グラフ（Depsgraph）の活用
 - 非同期処理とタイマー（ bpy.app.timers ）
 - イベントリスナーとリアルタイム更新
 - スレッドを使った処理（スレッドセーフな方法）
 - Blenderのファイルシステムを操作
 - 外部Pythonライブラリの利用
-

9.1 カスタム描画（OpenGL / GPU）

Blenderのビューポート上にカスタム描画するには、`bpy.types.SpaceView3D.draw_handler_add()`を使います。

9.1.1 シンプルな描画例

```
import bpy
import bgl
import blf

def draw_callback_px(self, context):
    blf.position(0, 100, 100, 0)
    blf.size(0, 20, 72)
    blf.draw(0, "Hello, Blender!")

class DrawOperator(bpy.types.Operator):
    """ビューポートにテキストを描画"""
    bl_idname = "view3d.draw_text"
    bl_label = "ビューポート描画"

    _handle = None

    def invoke(self, context, event):
        if self._handle is None:
            self._handle = bpy.types.SpaceView3D.draw_handler_add(
                draw_callback_px, (self, context), 'WINDOW', 'POST_PIXEL'
            )
        return {'RUNNING_MODAL'}

    def register():
        bpy.utils.register_class(DrawOperator)
```

```
bpy.utils.register_class(DrawOperator)

def unregister():
    bpy.utils.unregister_class(DrawOperator)

if __name__ == "__main__":
    register()
```

このスクリプトを実行すると、ビューポート上に「Hello, Blender!」と表示されます。

9.1.2 OpenGLの代替

Blender 2.8以降では、従来の `bg1` の代わりに `gpu` モジュールを使用することが推奨されています。

9.2 Blenderの依存グラフ（Depsgraph）の活用

Depsgraph（依存グラフ） は、Blenderのオブジェクトの状態や変更履歴を管理する仕組みです。

9.2.1 Depsgraphを使った更新検知

```
import bpy

def depsgraph_update(scene):
    print("オブジェクトが変更されました")

bpy.app.handlers.depsgraph_update_post.append(depsgraph_update)
```

このコードを実行すると、**オブジェクトが変更されるたびにログが出力** されます。

9.3 非同期処理とタイマー

通常の `time.sleep()` はBlenderをフリーズさせてしまうため、**非同期処理** を行うには `bpy.app.timers` を使用します。

9.3.1 タイマーを使った非同期処理

```
import bpy

def my_timer():
    print("1秒ごとに実行")
    return 1.0 # 1秒後に再実行

bpy.app.timers.register(my_timer)
```

このコードを実行すると、**1秒ごとにコンソールにログが表示** されます。

9.4 イベントリスナーとリアルタイム更新

オブジェクトが変更されたときにリアルタイムで処理を実行する場合、イベントリスナーを活用します。

9.4.1 イベントリスナーの作成

```
import bpy

def object_update(self, context):
    print(f"オブジェクトの位置が変更されました: {self.location}")

bpy.types.Object.location = bpy.props.FloatVectorProperty(update=object_update)
```

このコードを実行すると、**オブジェクトの位置が変更されるたびにログが出力** されます。

9.5 スレッドを使った処理

BlenderのPythonスクリプトは基本的に **シングルスレッド** ですが、計算が重い処理を並行して実行することも可能です。

9.5.1 スレッドを使った処理

```
import bpy
import threading
import time

def heavy_computation():
    time.sleep(5)
    print("重い計算が完了しました")

thread = threading.Thread(target=heavy_computation)
thread.start()
```

Blenderのデータ (**bpy.data** など) を別スレッドから直接変更するのはNGなので、結果はメインスレッドに戻して適用する必要があります。

9.6 Blenderのファイルシステムを操作

Blenderでは、スクリプトから **.blend** ファイルを読み書きできます。

9.6.1 **.blend** ファイルを開く

```
import bpy

bpy.ops.wm.open_mainfile(filepath="/path/to/your.blend")
```

9.6.2 .blend ファイルを保存

```
bpy.ops.wm.save_mainfile(filepath="/path/to/save.blend")
```

9.6.3 外部ファイルを読み込む

```
import json

with open("/path/to/config.json", "r") as f:
    data = json.load(f)
    print(data)
```

9.7 外部Pythonライブラリの利用

Blenderは **標準のPython環境** で動作しているため、**pip** を使って外部ライブラリを導入できます。

9.7.1 外部ライブラリのインストール

```
import subprocess
import sys

def install_package(package):
    subprocess.check_call([sys.executable, "-m", "pip", "install", package])

install_package("numpy")
```

このコードを実行すると、BlenderのPython環境に **numpy** がインストールされます。

9.7.2 NumPyを使った処理

```
import numpy as np

array = np.array([1, 2, 3])
print(array * 2)
```

Blenderでのデータ処理にNumPyを活用すると、**数値計算が高速化** できます。

9.8 まとめ

この章では、Blenderアドオン開発の高度なテクニックを学びました。

- カスタム描画（OpenGL / GPU）
 - 依存グラフ（Depsgraph）の活用
 - 非同期処理とタイマー（ bpy.app.timers ）
 - イベントリスナーを使ったリアルタイム更新
 - スレッドを使った並列処理
 - Blenderのファイル操作
 - 外部ライブラリの活用
-

9.9 次回予告

次回は「Blenderアドオン開発の最適なワークフロー」を解説します！
開発の効率を最大化する方法や、デバッグ・テストの手法を詳しく見ていきましょう！

第10章：Blenderアドオン開発の最適なワークフロー

前回はBlenderアドオン開発の高度なテクニックを学びました。
今回は、効率的な開発ワークフローとデバッグ・テストの手法を解説します。

アドオン開発では、開発効率を最大化し、バグを減らすことが重要です。
適切な開発環境を整え、デバッグ・テストを行うことで、高品質なアドオンを開発できます。

この章では、以下の内容を学びます。

- 開発環境のセットアップ
 - 外部エディタ（VS Code / PyCharm）の活用
 - Blenderのスクリプトリロード機能
 - デバッグ手法（エラーハンドリング / ログ出力）
 - ユニットテストの導入
 - アドオンの最適化
 - コード管理（GitHub / バージョン管理）
-

10.1 開発環境のセットアップ

Blenderのスクリプトエディタでもアドオン開発はできますが、
外部エディタを使用することで開発効率を大幅に向上させることができます。

10.1.1 推奨ツール

ツール	説明
VS Code	軽量で拡張機能が豊富。おすすめの開発環境
PyCharm	高機能なPython IDE。補完機能が強力
GitHub	コードのバージョン管理に必須
Blender Console	Blender内のPythonコンソールで動作確認
Blender System Console	エラーログや print() の出力を確認

10.2 外部エディタ（VS Code / PyCharm）の活用

10.2.1 VS Codeのセットアップ

1. VS Codeをインストール（[ダウンロード](#)）
2. 「Python」拡張機能をインストール
3. BlenderのPython環境を設定
 - Blenderの `python` を VS Code に認識させる
 - 例えば、Linuxの場合:

```
/path/to/blender/python/bin/python3 -m pip install --upgrade pip
```

4. VS CodeでBlenderのスクリプトフォルダを開く
 - Blenderのアドオンフォルダに直接編集できるようにする

10.3 Blenderのスクリプトリロード機能

Blenderでアドオンを編集するたびに **Blenderを再起動する**のは非効率です。
そのため、**スクリプトのリロード機能**を活用しましょう。

10.3.1 手動リロード

Blenderのアドオン設定画面で「アドオンを無効化 → 再度有効化」すると、スクリプトが再読み込まれます。

10.3.2 自動リロード

以下のコードを実行すると、アドオンのスクリプトを即座にリロードできます。

```
import bpy
bpy.ops.script.reload()
```

10.4 デバッグ手法

10.4.1 エラーハンドリング

エラー発生時にクラッシュを防ぐため、**例外処理（try-except）**を適切に使いましょう。

```
try:
    obj = bpy.context.object
    print(obj.name)
except AttributeError:
    print("オブジェクトが選択されていません")
```

10.4.2 ログ出力

エラーメッセージや変数の値を確認するため、**print()** や **report()** を活用 しましょう。

```
print("デバッグメッセージ")
bpy.context.window_manager.report({'INFO'}, "処理が完了しました")
```

10.4.3 System Console の活用

- **Windows:** Window > Toggle System Console
- **Mac/Linux:** Blenderをターミナルから起動するとログが表示される

10.5 ユニットテストの導入

アドオンの品質を向上させるため、**ユニットテスト（自動テスト）** を導入すると便利です。

10.5.1 Blenderのスクリプトでテスト

以下のコードは、オブジェクトの作成が正しく機能するかテストします。

```
import bpy

def test_create_object():
    bpy.ops.mesh.primitive_cube_add()
    obj = bpy.context.object
    assert obj is not None
    assert obj.type == 'MESH'
    print("テスト成功")

test_create_object()
```

Pythonの **unittest** モジュールを使って **テストスクリプトを実行** することもできます。

10.6 アドオンの最適化

Blenderアドオンは、**処理の速度とメモリ使用量を最適化** することが重要です。

10.6.1 高速なループ処理

bpy.ops は遅いため、**直接データを変更する方法が推奨** されます。

```
import bpy

# 遅い方法 ( bpy.ops )
```

```
bpy.ops.object.select_all(action='SELECT')

# 高速な方法(直接変更)
for obj in bpy.data.objects:
    obj.select_set(True)
```

10.6.2 メモリ管理

一時的なデータを作成したら、不要になったら削除することが重要です。

```
import bpy

# 一時オブジェクトを作成
bpy.ops.mesh.primitive_cube_add()
temp_obj = bpy.context.object

# 不要になったら削除
bpy.data.objects.remove(temp_obj)
```

10.7 コード管理（GitHub / バージョン管理）

アドオンのコードを適切に管理するために、GitHubやGitを活用 しましょう。

10.7.1 Gitの基本コマンド

```
git init # Gitリポジトリを作成
git add . # すべての変更を追加
git commit -m "初回コミット" # コミット
git push origin main # リモートにプッシュ
```

GitHubにコードをアップロードしておくと、過去のバージョンに戻せる ので安全です。

10.8 まとめ

この章では、Blenderアドオン開発の最適なワークフローについて学びました。

- VS CodeやPyCharmで効率的に開発
- Blenderのスクリプトリロード機能を活用
- エラーハンドリングとデバッグ手法
- ユニットテストを導入して品質向上
- アドオンの最適化で高速化
- GitHubを使ったバージョン管理

これらを活用することで、開発スピードを向上させ、バグの少ないアドオンを作成 できます。

10.9 次回予告

次回は「Blenderアドオン開発の総まとめ」を解説します！

これまで学んだ内容を整理し、実際のプロジェクトに活かす方法を考えていきましょう！

第11章：Blenderアドオン開発の総まとめ

これまでの章では、Blenderアドオンの開発手法を基礎から高度なテクニック、配布・収益化まで詳しく解説してきました。

この最終章では、学んだ内容を整理し、実際のプロジェクトに活かすためのロードマップを示します。

11.1 これまで学んだことの振り返り

1～5章：基礎

- アドオンの基本構造 (`__init__.py` と `register()` / `unregister()`)
- オペレーターの作成 (`bpy.types.Operator` を活用)
- パネルとメニューの追加 (`bpy.types.Panel` / `bpy.types.Menu`)
- Blenderのデータ構造とカスタムプロパティ
- Blenderのプリファレンスに設定を保存・読み込み

6～7章：配布・公開

- アドオンのパッケージ化 (ZIP形式)
- GitHubでの無料公開
- Blender Market / Gumroad / BOOTH での販売
- Blender公式リポジトリへの登録方法

8～10章：収益化・高度な開発

- 有料アドオンの価格設定とマーケティング
 - OpenGL / Depsgraph / 非同期処理を活用
 - 開発ワークフローの最適化 (VS Code / GitHub / 自動リロード)
-

11.2 実際のアドオン開発フロー

では、これまでの知識を活かして、実際の開発プロセスを整理してみましょう。

ステップ1：アイデアを決める

まず、どんなアドオンを作るのかを明確にする必要があります。

アイデアの出し方

- 「普段の作業で不便だと感じること」をリストアップ
- Blenderの既存機能を強化できるポイントを探す
- 他のアドオンと差別化できる要素を考える

例えば、以下のようなアイデアがあります。

アドオンの種類	ターゲットユーザー	例
ワークフロー支援	一般ユーザー	モデリング補助、レンダリング設定管理
リギングツール	アニメーター	自動リグ、ポーズライブリ拡張
VFX / シミュレーション	CGアーティスト	カスタムパーティクル、爆発エフェクト
エクスポートツール	ゲーム開発者	glTF / FBX エクスポートのカスタマイズ

ステップ2：プロトタイプを作る

アイデアが決まったら、まずは **最小限の機能（MVP）** を作ってみましょう。

「いきなり完成版を目指す」のではなく、まず動くものを作るのが重要です。

プロトタイプ開発の手順

1. 最低限のオペレーター（ボタン）を作る
2. 基本機能を実装
3. 3DビューのUIに追加
4. データを保存・読み込み
5. スクリプトリロードでテストしながら改善

例えば、「オブジェクトをX軸に2m移動するアドオン」なら、以下のようにシンプルなコードから始めます。

```
import bpy

class MoveXOperator(bpy.types.Operator):
    """X軸に2m移動するオペレーター"""
    bl_idname = "object.move_x"
    bl_label = "X軸に2m移動"

    def execute(self, context):
        obj = context.object
        if obj:
            obj.location.x += 2
            self.report({'INFO'}, "オブジェクトを移動しました")
        else:
            self.report({'WARNING'}, "オブジェクトを選択してください")
        return {'FINISHED'}

    def register():
        bpy.utils.register_class(MoveXOperator)

    def unregister():
        bpy.utils.unregister_class(MoveXOperator)
```

```
if __name__ == "__main__":
    register()
```

ステップ3：コードを整理・拡張

基本的な機能が動作するようになったら、コードを整理し、拡張します。

- **operators.py** (オペレーター)
- **ui.py** (パネル・メニュー)
- **utils.py** (ヘルパー関数)
- **preferences.py** (設定管理)

```
my_addon/
├── __init__.py
└── operators.py
├── ui.py
├── preferences.py
├── utils.py
├── README.md
├── LICENSE
└── example.blend
```

ステップ4：テストとデバッグ

開発後、以下の方法で動作を確認しましょう。

手動テスト

1. Blenderのコンソールでエラーをチェック
2. 異常動作しないかテスト
3. Blenderを再起動して設定が保存されるか確認

自動テスト（スクリプトで確認）

```
def test_move_x():
    bpy.ops.mesh.primitive_cube_add()
    obj = bpy.context.object
    obj.location.x = 0
    bpy.ops.object.move_x()
    assert obj.location.x == 2
    print("テスト成功！")

test_move_x()
```

ステップ5：ドキュメントを作成

アドオンを配布する前に、**README.md** を作成して、使い方を明記 しましょう。

MyAddon

Blenderアドオン「MyAddon」は、オブジェクトをX軸に2m移動するツールです。

インストール方法

1. `my_addon.zip` をダウンロード
2. Blenderの「プリファレンス」>「アドオン」>「インストール」で追加
3. 「ツール」タブにボタンが追加されます

使い方

- 3Dビューでオブジェクトを選択
- 「X軸に2m移動」ボタンをクリック

ステップ6：配布・販売

公開方法を選びます。

公開方法	特徴	推奨用途
GitHub	無料、オープンソース向け	コミュニティ向け
Blender Market	有料販売、手数料30%	高品質なアドオン
Gumroad	手数料8%、簡単に販売可能	個人販売向け
BOOTH	日本向け、円で販売可能	国内ユーザー向け

ステップ7：アップデートとユーザー対応

アドオンは **公開後も継続的に改善** することが重要です。

アップデートの流れ

1. バグ修正・新機能追加
2. GitHub / Blender Marketでアップデート
3. リリースノートを公開
4. ユーザーフィードバックを反映

11.3 まとめ

この章では、アドオン開発の流れを総まとめ しました。

- アイデアを決める
- プロトタイプを作る
- コードを整理・拡張

- テスト・デバッグ
- ドキュメントを作成
- 配布・販売
- アップデートとユーザー対応

この流れを参考にしながら、**Blenderアドオン開発を実践**してみてください！

11.4 最後に

これで **Blenderアドオン開発の完全ガイド**が完了しました！

次は、実際にアドオンを開発・公開してみましょう！ 

第12章：Blenderアドオン開発を成功させるための戦略と今後の展望

これまでの章では、Blenderアドオンの開発方法、配布、収益化、最適なワークフローについて学びました。最終章では、**アドオン開発を長期的に成功させるための戦略と、今後の展望**について解説します。

この章では、以下の内容を取り上げます。

- Blenderアドオン市場の動向
 - 長期的な収益を生み出す方法
 - ユーザーコミュニティの構築
 - 繼続的なアップデートの重要性
 - AIとBlenderアドオンの未来
 - 今後の目標設定とキャリア戦略
-

12.1 Blenderアドオン市場の動向

Blenderのアドオン市場は**年々成長**しています。

特に、プロフェッショナル向けのアドオンや作業効率を向上させるツールの需要が高まっています。

12.1.1 どんなアドオンが売れているのか？

Blender Market や Gumroad で人気のアドオンを分析すると、以下のカテゴリが特に注目されています。

カテゴリ	例	価格帯
モデリングツール	HardOps, BoxCutter	\$10～\$50
アニメーション支援	Auto-Rig Pro	\$20～\$100
レンダリング補助	EEVEE Ultimate Shader Pack	\$10～\$40
ゲーム開発向け	glTF Exporter	\$15～\$50
ワークフロー自動化	SimpleBake, SpeedRetopo	\$20～\$80

Blenderの進化に伴い、新しいツールの需要が生まれています。

市場のニーズを調査し、独自の価値を持つアドオンを開発することが成功のカギになります。

12.2 長期的な収益を生み出す方法

アドオンを1つ開発して終わりではなく、**長期的に収益を生み出す仕組み**を作ることが重要です。

12.2.1 継続的なアップデート

- 定期的に新機能を追加し、**ユーザーに価値を提供し続ける**
- Blenderのアップデートに対応し、常に動作する状態を維持

12.2.2 有料アップデートと拡張パック

- 基本機能を **\$10～\$30** で提供し、高度な機能は別売り (**\$50～\$100**)
- 「無料アドオン+プレミアム版（Pro）」という形も有効

12.2.3 月額課金モデル（サブスクリプション）

- PatreonやFanboxを活用し、**毎月の収益を安定化**
- アドオンの定期アップデートや限定コンテンツを提供

12.3 ユーザーコミュニティの構築

成功しているアドオン開発者は、**コミュニティを大切にしている**ことが共通しています。

12.3.1 Discordサーバーの活用

- ユーザー同士が交流できる場を提供する
- バグ報告やフィードバックを直接受け取れる

12.3.2 SNSでの情報発信

- Twitter, YouTube, Blender Artists** でアドオンを宣伝
- チュートリアル動画を作成し、アドオンの価値を伝える

12.3.3 無料ユーザーも大切にする

- 無料アドオンを配布し、**ブランド認知度を上げる**
- 一部機能を有料化することで収益を確保

12.4 継続的なアップデートの重要性

12.4.1 Blenderのバージョンアップ対応

- Blender 3.x → 4.x** では、APIの変更が発生するため、**定期的なメンテナンスが必要**
- 事前に **Blenderの開発版（Alpha, Beta）** をチェックし、動作確認

12.4.2 ユーザーからのフィードバックを活用

- GitHub Issues や Blender Market のレビューを参考に機能改善
- 「**ユーザーが欲しい機能**」を取り入れることで、売上を伸ばせる

12.5 AIとBlenderアドオンの未来

最近では、AI技術を活用したアドオンも登場しています。

12.5.1 AIを活用した新しいアドオン

- AIによるモデリング支援
- 画像から3Dメッシュを自動生成
- AIリギングアシスタント

12.5.2 ComfyUI や ControlNetとの連携

- AI画像生成（Stable Diffusion）とBlenderの連携
- AIによるマテリアル自動生成

今後、AI技術を取り入れたアドオン開発が大きなトレンドになる可能性があります。

12.6 今後の目標設定とキャリア戦略

アドオン開発を続けることで、個人のキャリアにも大きな影響を与えることができます。

12.6.1 フリーランスとしての活動

- オリジナルアドオンの販売で生計を立てる
- 企業向けにカスタムアドオンを開発する
- YouTube / Udemy でアドオン開発講座を展開

12.6.2 ゲーム・VFX業界への展開

- Blenderを活用するゲームスタジオ向けにツール開発
- 映画・アニメーション制作のワークフロー改善

12.6.3 自分だけのブランドを作る

- 「このアドオンといえば○○さん！」というブランドを確立
- オリジナルキャラクターやIPと組み合わせてアドオンを展開

Blenderアドオン開発は、単なる趣味にとどまらず、収益を生み出すビジネスやキャリアの武器になる可能性を秘めています。

12.7 まとめ

この章では、Blenderアドオン開発を成功させるための戦略と、今後の展望について解説しました。

- Blenderアドオン市場は成長中！
- 長期的な収益を生み出すためには、定期的なアップデートが重要
- ユーザーコミュニティを構築し、フィードバックを活用
- AI技術を取り入れることで、新しい可能性が広がる
- アドオン開発は、フリーランスや業界へのキャリアにつながる

12.8 最後に

これで **Blenderアドオン開発の完全ガイド** が完結しました！✿

ここまで学んだ知識を活かして、**自分だけのアドオンを開発し、世界中のBlenderユーザーに届けましょう！** ☺

Blenderの可能性は無限大です。

今こそ、あなたのアイデアを形にする時です！💡 ♡